BIOMEDICAL ENGINEERING DESIGN AND QUALITY ASSURANCE

EXERCICE MANUAL WITH PYTHON

Albert Guvenis
& Banu Andrea Ecaterina

Contents

Preface and Acknowledgements 3
1. Random Number generation 4
2. Monte Carlo Simulation for a Biomedical Engineering Application: Drug Efficacy 6
3. Monte Carlo Simulation for Design Optimization in Biomedical Engineering: Prosthetic Limb Design 8
4. Taguchi Method for Biomedical Instrument Design: A Python Example 10
5. Taguchi Design Example: Optimizing Signal-to-Noise Ratio in an Amplifier Circuit 13
6. Estimating Component Reliability and Predicting Failure Rates in Biomedical Instruments Using Python 17
7. Probabilistic Modeling for Root Cause Analysis in Biomedical Devices 20
8. Probabilistic Modeling for Preventive Maintenance Scheduling 24
9. Probabilistic Modeling for Bland-Altman Analysis in Python 27
10. Computing Repeatability for a Biomedical Measurement Instrument in Python 31
11. Computing Reproducibility for a Biomedical Measurement Instrument in Python 34
12. Computing Uncertainty in Biomedical Measurements Using Python and Error Propagation 37
13. Probabilistic Modeling for Sensor Calibration and Error Correction in Python 40
14. Probabilistic Modeling for Data Fusion in Python 43
15. Statistical Process Control for a Biomedical Engineering Application 47
16. Process Capability Analysis for a Biomedical Engineering Application 50
Further Reading 55
APPENDIX 56
A.1 List of basic Python commands 56
A.2 Essential NumPy Commands: A Quick Reference 58
A3. Probability Basics: 59
A4. Installing and running Python 60

Preface and Acknowledgements

Probabilistic modeling is a statistical approach that uses probability theory to represent uncertainty in data and models. It involves constructing mathematical models that describe the likelihood of different outcomes occurring. It has a vast number of applications in various fields such as engineering and economy.

This exercice book concentrates particularly on biomedical engineering design and quality assessment / assurance. Many engineering projects include computer modeling and simulation of the design. In this exercice book, we present examples of such methods based on probabilistic modeling. In particular an emphasis is given to Monte Carlo Simulations that can take the probabilistic nature of systems into account. Some engineering optimization methods are also presented that can help the designer to come up with the optimal parameter values that yield the best and most robust system. In engineering design, probabilistic models play also an important role in assessing measurement uncertainy. Reliability design is also an important issue in medical instrument design. Finally, these instruments must be maintained in time to continue their performance. Statistical process control techniques are often used as part of quality assurance for these systems.

The manual takes a practical approach to learning engineering design and quality assessment/ assurance through programming exercices with Python. Practical exercices are presented with the Python code along with the results. At the end of the book, there are suggestions for a number of further sources on probabilistic modeling.

An appendix is included that gives a short review of probability and Python basics. Installation tips are also included.

In our era of building “digital twins”, it is highly important to model and simulate systems in silico in order to design and optimize instruments, understand the mechanisms of normal functioning and disease, characterize diseases and patients using imaging and other non-invasive measurment methods, to perform therapy planning and design and optimize therapeutic methods and instruments and predict therapy responses for precision medicine. The goal is to diagnose and treat each patient in time and in the right way, reduce side effects, risks, errors and trials and errors while mazximizing efficacy.

The probabilistic modeling approach opens the way to have a holistic treatment of biomedical engineering, encompassing both quality assurance and precision medicine.

At the end of chapters, we also included some references from the research work of our group in these areas.

Acknowledgements:

The development, manual testing of the codes and their updates have been performed by Banu Andrea Ecaterina during her ERASMUS internship in the summer of 2025.

1. Random Number generation

Principles

There are two main kinds of random number generators: True Random Number Generators (TRNGs) and Pseudo-Random Number Generators (PRNGs). A seed value and a math approach that always gives the same answer are used by the most common type of random number generator (PRNG). The algorithm utilizes the first number as the seed for the next round. This makes a string of integers that seems random and passes a lot of tests. But the numbers aren't really random because they are made with a set algorithm that can be predicted, thus they will show up over and over again. The Mersenne Twister and the Linear Congruential Generator are examples of PRNGs that are good enough for most everyday jobs, such playing video games or running simulations that need to be fast and work over and over.

People use True Random Number Generators (TRNGs) for things like encryption and security that need to be completely random. They need to collect entropy, which is data from physical events outside the computer system that can't be predicted. This makes the numbers that come out look rather random. Some things that cause entropy are noise in the air, circuits that get hotter, or the exact time that someone types or movements their mouse. This is a truly random output since it turns physical data into a digital sequence of bits. You can utilize TRNGs to produce a safe, random seed that you then give to a quicker PRNG to make things function better. This makes a mixed system called a Cryptographically Secure PRNG (CSPRNG).

Code

```
import random

# Generate a random integer between 1 and 100
random_integer = random.randint(1, 100)
print("Random integer:", random_integer)

# Generate a random floating-point number between 0 and 1
random_float = random.random()
print("Random float:", random_float)

# Generate a random floating-point number between 5 and 10
random_float_range = random.uniform(5, 10)
print("Random float between 5 and 10:", random_float_range)

# Generate a random choice from a list
my_list = ["apple", "banana", "orange"]
random_choice = random.choice(my_list)
print("Random choice from the list:", random_choice)

# Generate a random sample of 3 elements from a list
random_sample = random.sample(my_list, 3)
print("Random sample of 3 elements:", random_sample)

# Seed the random number generator for reproducible results
random.seed(42)
random_number = random.randint(1, 100)
```

print("Random number with seed 42:", random_number)

Screen with results

References

Knuth, D.E. (1998) The art of computer programming, volume 2: Seminumerical algorithms. 3rd edn. Reading, Massachusetts: Addison-Wesley.

2. Monte Carlo Simulation for a Biomedical Engineering Application: Drug Efficacy

Principles

The Monte Carlo simulation is a computational method that models the probability of a wide range of outcomes in a complex system. It is based on repeated random sampling. Instead of using fixed input values, it assigns probability distributions (like the normal distribution or uniform distribution) to the uncertain variables in a mathematical model. The simulation then runs iterations and each time it picks a random value for the input variables from their specified distribution. The method then produces a distribution of possible outcomes. This way, it produces not just the most likely result, but also the full spectrum of potential risks and rewards. This makes it a powerful tool for forecasting and decision-making in engineering, and science.

Example

Problem: We want to simulate the effectiveness of a new drug in treating a disease. As a start, we have estimates of the drug's success rate and the standard deviation of the success rate. (For a realistic model, we may have a multifactorial model that relates the success rate to the factors). We want to estimate the drug's overall efficacy by the use Monte Carlo method. Below is the code for the simple intoductory case.

Code:

```
import numpy as np
# Define parameters
success_rate = 0.7 # Estimated success rate
std_dev = 0.05 # Standard deviation of success rate
num_simulations = 10000
# Run simulations
results = []
for _ in range(num_simulations):
    # Sample from a normal distribution
    simulated_success_rate = np.random.normal(success_rate, std_dev)
    results.append(simulated_success_rate)
# Calculate mean success rate from simulations
mean_success_rate = np.mean(results)

print("Estimated mean success rate:", mean_success_rate)
```

Screen with results

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
Random integer: 13
Random float: 0.07824560654550383
Random float between 5 and 10: 5.492041524043982
Random choice from the list: orange
Random sample of 3 elements: ['apple', 'banana', 'orange']
Random number with seed 42: 82
PS C:\Users\PC\Documents\Python> & C:/Users/PC/Documents/Python/.venv/Scripts/python.exe "c:/Users/PC/Documents/Python/Monte Carlo Simulation.py"
Estimated mean success rate: 0.7003025439228482
PS C:\Users\PC\Documents\Python> & C:/Users/PC/Documents/Python/.venv/Scripts/python.exe "c:/Users/PC/Documents/Python/MCSProsthetic Limb Design.PY"
No valid designs found
PS C:\Users\PC\Documents\Python>
Ln 63, Col 36 Spaces: 4 UTF-8 CRLF {} Python

Additional Considerations:

Confidence intervals: To estimate the range of possible success rates, calculate confidence intervals around the mean.
Sensitivity analysis: Vary the input parameters to assess how the results change.
Multiple variables: If the drug's efficacy depends on multiple factors, incorporate them into the simulation.
Visualization: Create plots to visualize the distribution of simulated results.

References

1. Oral, A. and Guvenis, A., 2021. Improving the detectability of overactive glands in dual-phase parathyroid SPECT/CT: a Monte Carlo simulation study. Biomedical Physics & Engineering Express, 7(4), p.045023.
2. Talat, D. and Guvenis, A., 2015. Design and evaluation of a breast-specific collimator using response surface methodology and Monte Carlo simulations. IEEE Transactions on Nuclear Science, 63(1), pp.98-107.

3. Monte Carlo Simulation for Design Optimization in Biomedical Engineering: Prosthetic Limb Design

Problem: We want to design a prosthetic limb that is both lightweight and durable. We have constraints on weight and strength. We want to find the optimal combination of materials and dimensions to meet these constraints.

Code

```
import numpy as np
# Define parameters
num_simulations = 10000
weight_constraint = 2.5 # Maximum weight in kg
strength_constraint = 500 # Minimum strength in N

# Material properties
materials = ["aluminum", "titanium", "carbon_fiber"]
weights = [2.7, 4.5, 1.8] # Density in kg/m^3
strengths = [200, 400, 500] # Tensile strength in N/m^2

# Design parameters
dimensions = np.random.uniform(0.1, 0.5, size=(num_simulations, 3)) # Length, width, height in meters

# Run simulations
results = []
for i in range(num_simulations):
    material_index = np.random.randint(0, len(materials))
    weight = weights[material_index] * np.prod(dimensions[i])
    strength = strengths[material_index] * dimensions[i][0] * dimensions[i][1]
    results.append((weight, strength))
# Filter results based on constraints
valid_results = [result for result in results if result[0] <= weight_constraint and result[1] >= strength_constraint]

# Find optimal design
if valid_results:
    optimal_design = min(valid_results, key=lambda x: x[0])
    print("Optimal design:", optimal_design)
else:
    print("No valid designs found")
```

Screen with results

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
Random choice from the list: orange
Random sample of 3 elements: ['apple', 'banana', 'orange']
Random number with seed 42: 82
PS C:\Users\PC\Documents\Python> & C:/Users/PC/Documents/Python/.venv/Scripts/python.exe "c:/Users/PC/Documents/Python/Monte Carlo Simulation.py"
Estimated mean success rate: 0.7003025439228482
PS C:\Users\PC\Documents\Python> & C:/Users/PC/Documents/Python/.venv/Scripts/python.exe "c:/Users/PC/Documents/Python/MCSProsthetic Limb Design.PY"
No valid designs found
PS C:\Users\PC\Documents\Python> & C:/Users/PC/Documents/Python/.venv/Scripts/python.exe "c:/Users/PC/Documents/Python/MCSProsthetic Limb Design.PY"
Optimal design: (np.float64(0.06458500007094865), np.float64(52.30159471859095))
PS C:\Users\PC\Documents\Python>
Ln 27, Col 40 Spaces: 4 UTF-8 CRLF {} Python

Explanation:

Define parameters: Set the number of simulations, constraints, material properties, and design parameters.
Run simulations: Generate random designs and calculate their weight and strength.
Filter results: Keep only designs that meet the constraints.
Find optimal design: Find the design with the lowest weight among the valid designs.

Additional Considerations:

More complex designs: Incorporate additional design parameters and constraints.
Optimization algorithms: Consider using optimization algorithms (e.g., genetic algorithms) for more complex problems.
Uncertainty analysis: Assess the sensitivity of the results to variations in input parameters.

This example demonstrates how Monte Carlo simulations can be used for design optimization in biomedical engineering.

4. Taguchi Method for Biomedical Instrument Design: A Python Example

Principles

The Taguchi Method is a systematic statistical way to improve quality by lowering variance and making a solid design that doesn't vary based on outside elements that can't be controlled (called "noise factors"). It was developed by Japanese engineer Genichi Taguchi. This way of thinking states that quality should be integrated into the product instead than being examined while it is being made. This is because it works best to eliminate discrepancies. The Taguchi Loss Function is a very important idea. It is a quadratic equation that demonstrates how much money society loses when the quality of a product changes, even if it is still acceptable. So, the goal isn't just to meet the requirements; it's to strike the target every time. This will reduce losses, raise quality, and make customers happier.

The Taguchi method needs special Design of Experiments (DOE) methodologies to develop a solid design. Using orthogonal arrays to find out how different design parameters and their relationships affect quality with fewer tests is one technique to do this. The Signal-to-Noise (S/N) ratio is what we use to look at the experimental data. It displays how much of the "signal" (the expected performance) there is compared to the "noise" (the unwanted change). You need to get this ratio as high as possible to find the ideal settings for the control variables (the things you can modify) that maintain the product's performance constant and less impacted by things like temperature or material wear and tear. The procedure has three steps: designing the system, designing the parameters, and designing the tolerances.

Example

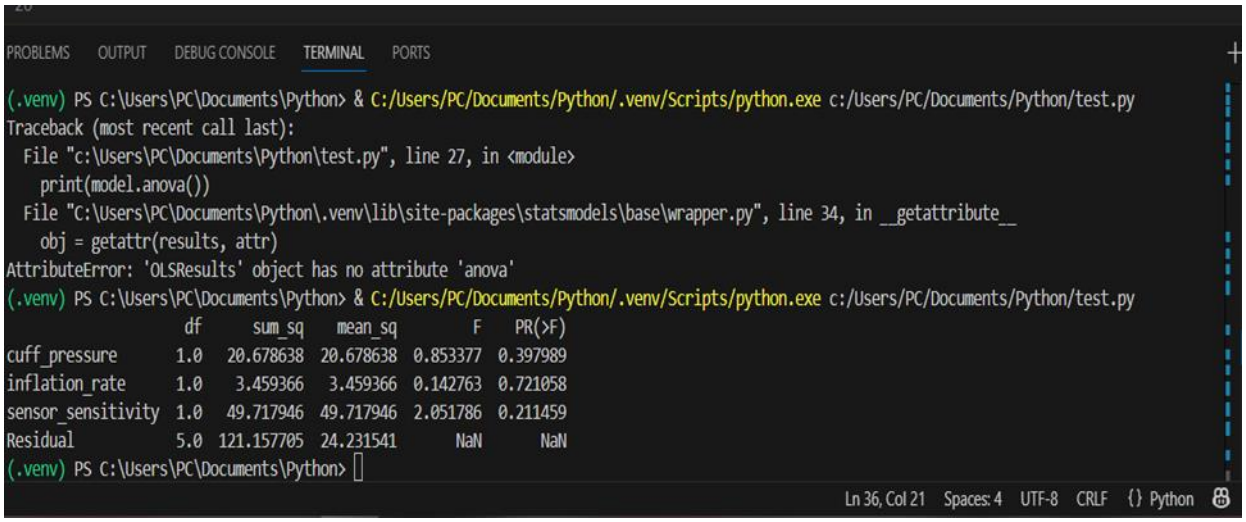
Problem: We want to design a portable blood pressure monitor that is accurate, reliable, and easy to use. We have three factors that we believe affect the accuracy of the device: cuff pressure, inflation rate, and sensor sensitivity.

Code:

```
import numpy as np
import pandas as pd
import statsmodels.api as sm
import statsmodels.formula.api as smf
from statsmodels.stats.anova import anova_lm

# Define factors and levels
factors = ["cuff_pressure", "inflation_rate", "sensor_sensitivity"]
levels = [2, 3, 3]

# Create orthogonal array
orthogonal_array = np.array([
    [1, 1, 1],
    [1, 2, 2],
    [1, 3, 3],
    [2, 1, 2],
```

```
    [2, 2, 3],
    [2, 3, 1],
    [3, 1, 3],
    [3, 2, 1],
    [3, 3, 2]
])

# Generate experimental data (replace with real data)
data = np.random.normal(120, 5, 9)

# Create DataFrame
df = pd.DataFrame(orthogonal_array, columns=factors)
df["accuracy"] = data

# Fit ANOVA model
model = smf.ols(formula="accuracy ~ cuff_pressure + inflation_rate + sensor_sensitivity",
data=df).fit()

# Print ANOVA table
anova_results = anova_lm(model)
print(anova_results)
```

Screen with results


PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
(.venv) PS C:\Users\PC\Documents\Python> & C:/Users/PC/Documents/Python/.venv/Scripts/python.exe c:/Users/PC/Documents/Python/test.py
Traceback (most recent call last):
File "c:\Users\PC\Documents\Python\test.py", line 27, in <module>
print(model.anova())
File "C:\Users\PC\Documents\Python\.venv\lib\site-packages\statsmodels\base\wrapper.py", line 34, in __getattribute__
obj = getattr(results, attr)
AttributeError: 'OLSResults' object has no attribute 'anova'
(.venv) PS C:\Users\PC\Documents\Python> & C:/Users/PC/Documents/Python/.venv/Scripts/python.exe c:/Users/PC/Documents/Python/test.py
df sum_sq mean_sq F PR(>F)
cuff_pressure 1.0 20.678638 20.678638 0.853377 0.397989
inflation_rate 1.0 3.459366 3.459366 0.142763 0.721058
sensor_sensitivity 1.0 49.717946 49.717946 2.051786 0.211459
Residual 5.0 121.157705 24.231541 NaN NaN
(.venv) PS C:\Users\PC\Documents\Python>
Ln 36, Col 21 Spaces: 4 UTF-8 CRLF {} Python

Explanation:

Define factors and levels: We define the factors and their corresponding levels.
Create orthogonal array: We create an orthogonal array using a standard design, such as the L9 array.
Generate experimental data: We collect data for each combination of factor levels.
Create DataFrame: We create a Pandas DataFrame to store the data.
Fit ANOVA model: We fit an ANOVA model to analyze the effects of the factors on the response

variable (accuracy).

Additional Considerations:

Signal-to-noise ratio: Calculate the signal-to-noise ratio to assess the relative importance of each factor.
Optimization: Use the results of the Taguchi analysis to optimize the design parameters.
Robust design: Consider using robust design techniques to minimize the effects of noise factors.

This example demonstrates how the Taguchi method can be used to optimize the design of a biomedical instrument. By systematically varying the factors and analyzing the results, we can identify the optimal combination of design parameters to achieve the desired performance.

References

Ross, P. J., 1996. Taguchi Techniques for Quality Engineering: Loss Function, Orthogonal Experiments, Parameter and Tolerance Design. 2nd edn. New York:

5. Taguchi Design Example: Optimizing Signal-to-Noise Ratio in an Amplifier Circuit

Scenario: An engineer is designing a low-noise amplifier circuit. The goal is to maximize the signal-to-noise ratio (SNR) at the output (response variable) while minimizing variations in SNR due to component tolerances and manufacturing variations.
Control Factors and Levels:
Factor A: Resistor Tolerance (%): Levels: 1% (tight tolerance), 5% (standard tolerance), 10% (loose tolerance)
Factor B: Capacitor Tolerance (%): Levels: 5% (standard tolerance), 10% (loose tolerance), 15% (wide tolerance)

Orthogonal Array Selection:
Similar to the previous example, we can use an L9 orthogonal array to evaluate all nine combinations of factor levels efficiently.
ExperimentFactor A (Resistor Tolerance)Factor B (Capacitor Tolerance)11% (Level 1)5% (Level 1)21% (Level 1)10% (Level 2)31% (Level 1)15% (Level 3)45% (Level 2)5% (Level 1)55% (Level 2)10% (Level 2)65% (Level 2)15% (Level 3)710% (Level 3)5% (Level 1)810% (Level 3)10% (Level 2)910% (Level 3)15% (Level 3)Conducting the Experiment:

For each experiment, the amplifier circuit is built with components having the corresponding tolerance levels for resistors and capacitors. The SNR at the output is then measured using a signal generator and a spectrum analyzer. The experiment is repeated for a statistically significant number of replicates (e.g., 3 replicates per experiment).

Data Analysis and S/N Ratio Calculation:
We will again use the "larger-the-better" loss function for the S/N ratio since we want to maximize the SNR. The S/N ratio can be calculated in decibels (dB) using the following formula:
S/N ratio (dB) = 10 log (signal power / noise power)
Mean SNR (dB) and S/N ratios for each experiment are calculated based on the replicates.

Identifying Optimal Settings:
Analyzing the S/N ratios for each factor level will reveal the combination that leads to the highest average S/N ratio. This combination represents the most robust component tolerances for achieving a consistently high SNR in the amplifier circuit, minimizing the impact of manufacturing variations. Additionally, ANOVA can be performed to assess the statistical significance of the impact of each factor on the S/N ratio.

This example showcases how the Taguchi method can be applied in electrical engineering to optimize circuit performance by strategically selecting component tolerances based on their influence on the desired output characteristic (SNR).
3.10 Response Surface Methodology (RSM) for Biomedical Instrument Design: A Python Example
Problem:

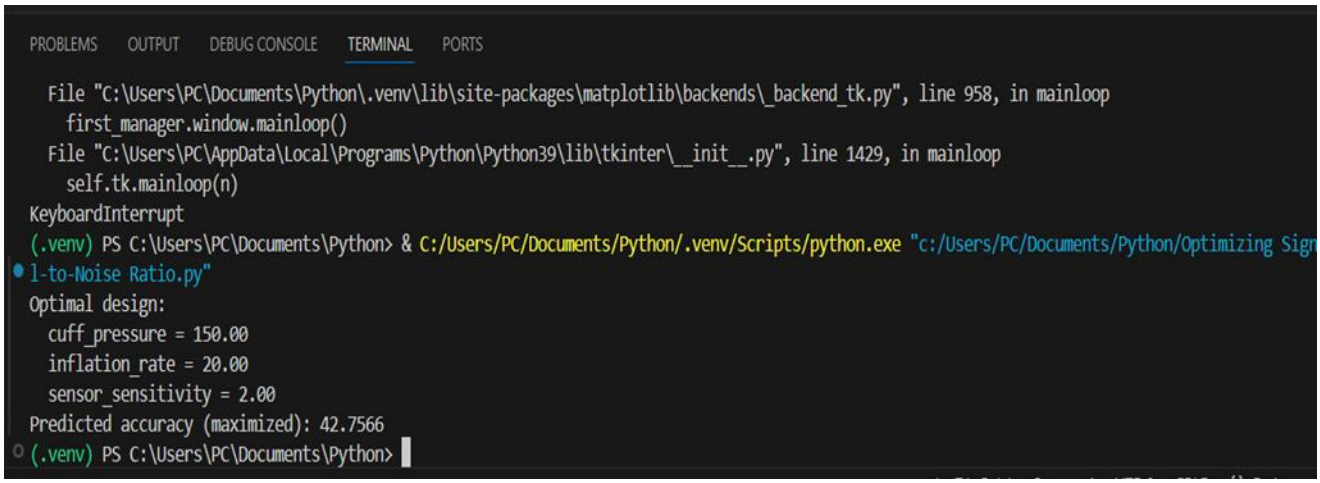
We want to optimize the design of a portable blood pressure monitor by adjusting three factors: cuff pressure, inflation rate, and sensor sensitivity. We aim to maximize accuracy while minimizing cost.

Code:

```
import numpy as np
import pandas as pd
import statsmodels.formula.api as sm
from scipy.optimize import minimize
# Define factors and their ranges
factors = ["cuff_pressure", "inflation_rate", "sensor_sensitivity"]
ranges = [(100, 150), (20, 40), (0.5, 2)]
# Simulate experimental data (e.g. 15 samples)
np.random.seed(42)
n_samples = 15
data = pd.DataFrame({
    "cuff_pressure": np.random.uniform(ranges[0][0], ranges[0][1], n_samples),
    "inflation_rate": np.random.uniform(ranges[1][0], ranges[1][1], n_samples),
    "sensor_sensitivity": np.random.uniform(ranges[2][0], ranges[2][1], n_samples),
})
# Simulate accuracy as a function of factors (quadratic relation + noise)
data["accuracy"] = (
    0.5 * data["cuff_pressure"]
    - 0.2 * data["inflation_rate"]
    + 1.5 * data["sensor_sensitivity"]
    - 0.01 * data["cuff_pressure"] * data["inflation_rate"]
    + 0.05 * data["inflation_rate"] * data["sensor_sensitivity"]
    - 0.3 * (data["sensor_sensitivity"] ** 2)
    + np.random.normal(0, 1, n_samples)  # noise
)
# Fit response surface model (linear + interactions + quadratic terms)
formula = (
    "accuracy ~ cuff_pressure + inflation_rate + sensor_sensitivity "
    "+ cuff_pressure:inflation_rate + cuff_pressure:sensor_sensitivity +
inflation_rate:sensor_sensitivity "
    "+ I(cuff_pressure**2) + I(inflation_rate**2) + I(sensor_sensitivity**2)"
)
model = sm.ols(formula=formula, data=data).fit()
# Define a simple cost function (increasing with cuff_pressure and sensor_sensitivity)
def cost_function(x):
    cuff_pressure, inflation_rate, sensor_sensitivity = x
    return 0.01 * cuff_pressure + 0.02 * sensor_sensitivity
# Objective function: maximize accuracy - cost
def objective_function(x):
    cuff_pressure, inflation_rate, sensor_sensitivity = x
    df_pred = pd.DataFrame([x], columns=factors)
    predicted_accuracy = model.predict(df_pred)[0]
    predicted_cost = cost_function(x)
    return -predicted_accuracy + predicted_cost  # minimize negative accuracy + cost
# Initial guess (mid-points)
x0 = [
    np.mean(ranges[0]),
    np.mean(ranges[1]),
```

```
    np.mean(ranges[2]),
]
# Bounds to respect the ranges of the factors
bounds = ranges
# Optimize
result = minimize(objective_function, x0=x0, bounds=bounds)
print("Optimal design:")
print(f"  cuff_pressure = {result.x[0]:.2f}")
print(f"  inflation_rate = {result.x[1]:.2f}")
print(f"  sensor_sensitivity = {result.x[2]:.2f}")
print(f"Predicted accuracy (maximized): {-result.fun:.4f}")
```

Screen with results:



Explanation:

Define factors and ranges: Specify the factors and their potential ranges.
Generate experimental design: Use a suitable experimental design (e.g., central composite design) to collect data.
Fit a response surface model: Fit a polynomial model to the experimental data to approximate the relationship between the factors and the response (accuracy).
Define objective function: Create a function that calculates the objective to be minimized (in this case, the negative of accuracy plus cost).
Optimize: Use a gradient-based optimization algorithm (e.g., SLSQP) to find the optimal values of the factors.

Additional Considerations:

Constraints: Incorporate constraints on the factors (e.g., physical limitations, cost constraints) into the optimization problem.
Uncertainty: Consider uncertainty in the model and experimental data by using probabilistic models or sensitivity analysis.
Multiple objectives: If there are multiple objectives (e.g., accuracy and power consumption), use multi-objective optimization techniques.

By following these steps, you can effectively use RSM to optimize the design of biomedical instruments and improve their performance.

6. Estimating Component Reliability and Predicting Failure Rates in Biomedical Instruments Using Python

Principles

The Weibull distribution is the most frequent statistical model used in reliability engineering to look at life data and forecast how long a product will survive. The reason for this is that it is so adaptable. The Weibull function can be changed to meet practically any failure pattern that real-world parts show, unlike the normal or exponential distributions. This makes it beneficial for every stage of a product's life cycle. The main use of this is to find out the chance of failure over time. This helps engineers find things like the Mean Time To Failure (MTTF), B-life (the time when a certain percentage of the population is expected to fail), and most importantly, the instantaneous failure rate through its parameters. Weibull Analysis is a key aspect in figuring out how long warranties should last, how to schedule preventative maintenance, and how to make sensible design choices.

The Weibull distribution is quite versatile because of its two primary parameters: the shape parameter (β) and the scale parameter (η). The form parameter, which is also called the Weibull slope, is a number without any units that shows what kind of failure mode it is:

The failure rate is getting reduced when $\beta<1$. This is what usually happens when anything goes wrong early in life or when a baby dies (for example, weak pieces failing during burn-in).

The failure rate stays the same when $\beta=1$, which means that failures happen at random. This means that the Weibull is the same as the exponential distribution (the product's "useful life").

When $\beta>1$, the failure rate is rising, which suggests that wear-out failures are occurring, such as fatigue or corrosion.
The scale parameter (η), which is also called the typical life, tells us when most failures occurs. This is the time when 63.2% of the population is expected to have died, which provides us an estimate of how long the product will last. By properly fitting life data to these standards, reliability engineers may effectively manage product risk and forecast how reliable a product will be in the future.

Example

Understanding the Problem:
In biomedical instruments, component reliability is crucial for ensuring device safety and efficacy. We can use probabilistic models to estimate component reliability and predict failure rates.

Code

```
import numpy as np
import scipy.stats as stats
from scipy.special import gamma # funcția gamma corectă
# Define component parameters
component_name = "capacitor"
mean_lifetime = 10000 # Hours (not used directly here)
std_dev = 2000 # Hours (not used directly here)
# Weibull distribution parameters
```

```
shape_param = 2.5 # also called beta
scale_param = 10000 # also called eta
# Calculate reliability at a specific time
time = 5000 # Hours
reliability = 1 - stats.weibull_min.cdf(time, shape_param, scale=scale_param)
print("Reliability at 5000 hours:", reliability)

# Predict failure rate at a specific time
failure_rate = stats.weibull_min.pdf(time, shape_param, scale=scale_param) / reliability
print("Failure rate at 5000 hours:", failure_rate)

# Calculate mean time to failure (MTTF)
mttf = scale_param * gamma(1 + 1 / shape_param)
print("Mean time to failure (MTTF):", mttf)
```

Screen with results:

PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
Reliability at 5000 hours: 1.0
Failure rate at 5000 hours: 0.0
Traceback (most recent call last):
File "c:\Users\PC\Documents\Python\Estimating Component Reliability and Predicting Failure Rates in Biomedical Instruments Using Python.py", line 45, in <module>
mttf = scale_param * stats.gamma.gamma(1 + 1 / shape_param)
AttributeError: 'gamma_gen' object has no attribute 'gamma'
(.venv) PS C:\Users\PC\Documents\Python> & C:/Users/PC/Documents/Python/.venv/Scripts/python.exe "c:/Users/PC/Documents/Python/Estimating Component Reliability and Predicting Failure Rates in Biomedical Instruments Using Python.py"
Reliability at 5000 hours: 0.8379668855787558
Failure rate at 5000 hours: 8.838834764831844e-05
Mean time to failure (MTTF): 8872.638175030752
(.venv) PS C:\Users\PC\Documents\Python>
Ln 26, Col 1
Spaces: 4
UTF-8
CRLF
{} Python

Explanation:

Define component parameters: Specify the component name, mean lifetime, and standard deviation.
Choose a distribution: For many components, the Weibull distribution is a suitable model.
Calculate reliability: Use the Weibull distribution's cumulative distribution function (CDF) to calculate the probability of the component surviving beyond a given time.
Calculate failure rate: Use the Weibull distribution's probability density function (PDF) to calculate the instantaneous failure rate at a specific time.
Calculate MTTF: Calculate the mean time to failure using the Weibull distribution's parameters.

Additional Considerations:

Data-driven approach: If you have historical failure data, you can fit a Weibull distribution to the

data to estimate the parameters.
Multiple failure modes: Consider multiple failure modes and use a mixture model to represent the overall reliability.
Environmental factors: Account for environmental factors (e.g., temperature, humidity) that can affect component reliability.
Maintenance strategies: Use reliability analysis to inform maintenance strategies and optimize device lifetime.
By using probabilistic modeling, you can estimate component reliability and predict failure rates, which is essential for ensuring the safety and efficacy of biomedical instruments.

References

Ebeling, CE 2019, An Introduction to Reliability and Maintainability Engineering, 3rd edn, Waveland Press, Long Grove.

7. Probabilistic Modeling for Root Cause Analysis in Biomedical Devices

Principles

Bayesian Hierarchical Model (BHM) is a robust probabilistic framework for the simultaneous analysis of reliability data collected from many origins, including numerous components, locations, or test batches, where the failure processes are analogous yet display heterogeneity. The model has several layers. The lowest level uses system-specific parameters and explanatory variables (factors), including operating temperature, stress level, or material attributes, to figure out how likely it is that a component would fail (for example, using a Weibull or log-normal distribution). The higher level of the hierarchy adds hyperparameters that presume that the parameters of each component come from the same population distribution. This lets information be "pooled" or shared between groups. This hierarchical structure is important because it takes into consideration both the uncertainty in each group's failure process and the differences between groups. This makes the estimations of the parameters related to the causes that cause failure more reliable.

Markov Chain Monte Carlo (MCMC) sampling is used to solve this complicated BHM and find the components that are most closely linked to failures. The posterior distribution, which shows what we think about the model parameters (including the factor effects) after seeing the failure data, can't usually be calculated analytically. Instead, MCMC algorithms like the Metropolis-Hastings or Gibbs Sampler are used to make a large number of parameter samples. These samples create a Markov Chain that finally reaches the true shape of the posterior distribution for each parameter in the model. An engineer can determine the following by looking at the marginal posterior distributions for the factor-related parameters (for example, the coefficients of a log-linear model for the failure rate): 1) the size of the factor's effect on reliability, and 2) the uncertainty of that effect (measured by credible intervals). Factors with posterior distributions distinctly separated from zero are regarded as the most significant predictors of failure.

Example:

Problem:
We want to identify the root causes of failures in a biomedical device. We have data on the device's operating conditions, component failures, and maintenance history.

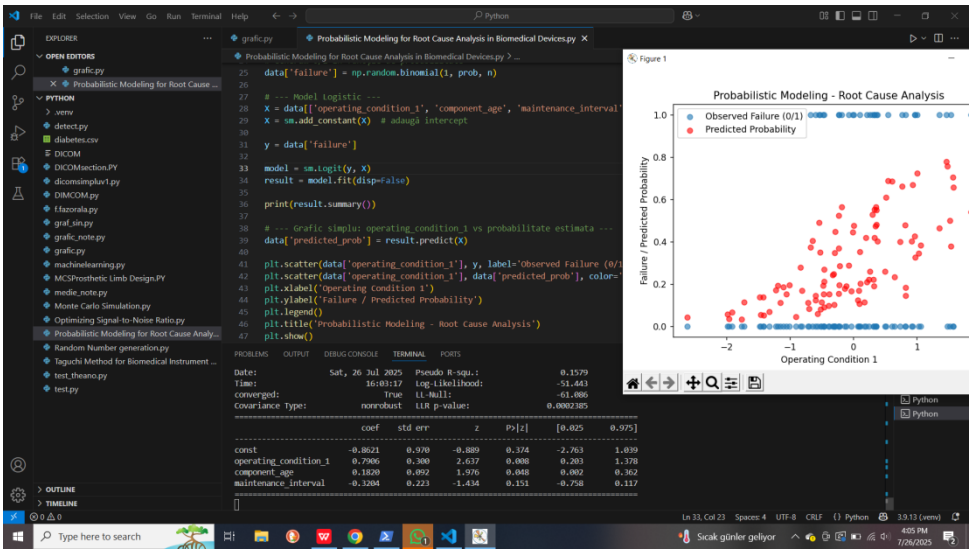
Code

```
import pandas as pd
import numpy as np
import statsmodels.api as sm
import matplotlib.pyplot as plt

# --- Artificial data (if you don't have a file) ---
np.random.seed(42)
n = 100
data = pd.DataFrame({
    'operating_condition_1': np.random.normal(0, 1, n),
    'component_age': np.random.uniform(0, 10, n),
    'maintenance_interval': np.random.uniform(1, 5, n)
```

```
})

# Create a true logistic relationship (probability)
logit_true = (
    -1.0
    + 0.8 * data['operating_condition_1']
    + 0.3 * data['component_age']
    - 0.5 * data['maintenance_interval']
)
prob = 1 / (1 + np.exp(-logit_true))

# Generate 0/1 based on probability
data['failure'] = np.random.binomial(1, prob, n)

# --- Logistic Regression Model ---
X = data[['operating_condition_1', 'component_age', 'maintenance_interval']]
X = sm.add_constant(X)  # add intercept
y = data['failure']
model = sm.Logit(y, X)
result = model.fit(disp=False)
print(result.summary())

# --- Simple plot: operating_condition_1 vs estimated probability ---
data['predicted_prob'] = result.predict(X)
plt.scatter(data['operating_condition_1'], y, label='Observed Failure (0/1)', alpha=0.6)
plt.scatter(data['operating_condition_1'], data['predicted_prob'], color='red', label='Predicted
Probability', alpha=0.6)
plt.xlabel('Operating Condition 1')
plt.ylabel('Failure / Predicted Probability')
plt.legend()
plt.title('Probabilistic Modeling - Root Cause Analysis')
plt.show()
```

Screen with results

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

(.venv) PS C:\Users\PC\Documents\Python> & C:/Users/PC/Documents/Python/.venv/Scripts/python.exe c:/Users/PC/Documents/Python/test.py
                           Logit Regression Results
==============================================================================
Dep. Variable:                failure   No. Observations:                  100
Model:                          Logit   Df Residuals:                       96
Method:                           MLE   Df Model:                            3
Date:                Mon, 25 Aug 2025   Pseudo R-squ.:                 0.1579
Time:                        21:16:00   Log-Likelihood:                -51.443
converged:                       True   LL-Null:                       -61.086
Covariance Type:            nonrobust   LLR p-value:                 0.0002385
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const         -0.8621      0.970     -0.889      0.374      -2.763       1.039
```



Explanation:

Load data: Load the data containing information on device operating conditions, component failures, and maintenance history.
Define probabilistic model: Define a probabilistic model using Bayesian hierarchical modeling. This model assumes that the failure probability is influenced by various factors (e.g., operating conditions, component age, maintenance interval).
Inference: Use Markov Chain Monte Carlo (MCMC) sampling to infer the posterior distribution of the model parameters.
Analyze results: Examine the posterior distributions of the model parameters to identify the factors that are most strongly associated with failures.

Abaei, M., Marseguerra, M., & Zio, E. (2018). Bayesian Hierarchical Methods for Modeling Electrical Grid Component Failures. arXiv preprint arXiv:2001.07597.

Additional Considerations:

Model complexity: The complexity of the model can be adjusted based on the available data and domain knowledge.
Prior distributions: The choice of prior distributions can influence the results.
Variable selection: Use techniques like variable selection or regularization to identify the most important factors.
Sensitivity analysis: Assess the sensitivity of the results to changes in the model parameters.
By using probabilistic modeling, we can identify the root causes of failures in biomedical devices and inform preventive maintenance strategies to improve device reliability.

References

1. Ebeling, C. E. 2019, An Introduction to Reliability and Maintainability Engineering, 3rd edn, Waveland Press, Long Grove.

2. O'Connor, P. D. T. & Kleyner, A. 2012, Practical Reliability Engineering, 5th edn, John Wiley & Sons, Ltd., Chichester.

3. Gelman, A., Carlin, J. B., Stern, H. S., Dunson, D. B., Vehtari, A. & Rubin, D. B. 2013, Bayesian Data Analysis, 3rd edn, CRC Press, Boca Raton.

4. Abaei, M., Marseguerra, M. & Zio, E. 2018, 'Bayesian Hierarchical Methods for Modeling Electrical Grid Component Failures', arXiv preprint arXiv:2001.07597, viewed 12 October 2025,

8. Probabilistic Modeling for Preventive Maintenance Scheduling

Example

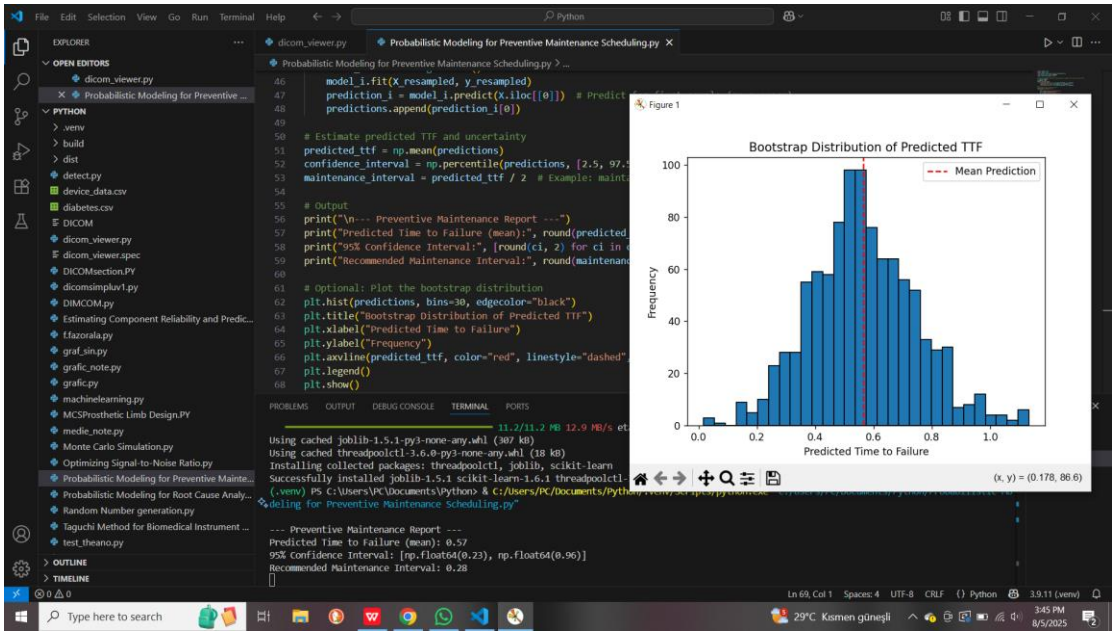
Problem:
We want to schedule preventive maintenance for a biomedical device to minimize downtime while ensuring optimal performance. We have data on the device's operating conditions, maintenance history, and failure modes.

Code:

```
import pandas as pd
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.utils import resample
import matplotlib.pyplot as plt

# Generate synthetic data for testing (you can replace this with your actual CSV)
np.random.seed(42)
n_samples = 100
data = pd.DataFrame({
"operating_condition_1": np.random.normal(0, 1, n_samples),
"component_age": np.random.normal(5, 2, n_samples),
"maintenance_interval": np.random.normal(10, 3, n_samples),
})
# Simulated time to failure (TTF)
data["time_to_failure"] = (
2.0 +
0.5 * data["operating_condition_1"] -
0.3 * data["component_age"] -
0.1 * data["maintenance_interval"] +
np.random.normal(0, 1, n_samples)
)
# Save to CSV (optional, or comment out if you already have one)
data.to_csv("device_data.csv", index=False)
# Load your data
data = pd.read_csv("device_data.csv")
# Features and target
X = data[["operating_condition_1", "component_age", "maintenance_interval"]]
y = data["time_to_failure"]
# Fit the base model
model = LinearRegression()
model.fit(X, y)
base_prediction = model.predict(X)
# Bootstrapping to estimate uncertainty
n_iterations = 1000
predictions = []
for i in range(n_iterations):
```

```
    X_resampled, y_resampled = resample(X, y)
    model_i = LinearRegression()
    model_i.fit(X_resampled, y_resampled)
    prediction_i = model_i.predict(X.iloc[[0]])  # Predict for first sample (or average)
    predictions.append(prediction_i[0])# Estimate predicted TTF and uncertainty
predicted_ttf = np.mean(predictions)
confidence_interval = np.percentile(predictions, [2.5, 97.5])
maintenance_interval = predicted_ttf / 2 # Example: maintain at half-life
# Output
print("\n--- Preventive Maintenance Report ---")
print("Predicted Time to Failure (mean):", round(predicted_ttf, 2))
print("95% Confidence Interval:", [round(ci, 2) for ci in confidence_interval])
print("Recommended Maintenance Interval:", round(maintenance_interval, 2))
# Optional: Plot the bootstrap distribution
plt.hist(predictions, bins=30, edgecolor="black")
plt.title("Bootstrap Distribution of Predicted TTF")
plt.xlabel("Predicted Time to Failure")
plt.ylabel("Frequency")
plt.axvline(predicted_ttf, color="red", linestyle="dashed", label="Mean Prediction")
plt.legend()
plt.show()
```

Screen with results:



Explanation:

Load data: Load the data containing information on device operating conditions, maintenance history, and failure data.
Define probabilistic model: Define a probabilistic model using Bayesian hierarchical modeling. This model assumes that the time to failure is influenced by various factors (e.g., operating conditions, component age, maintenance interval).
Inference: Use MCMC sampling to infer the posterior distribution of the model parameters.

Calculate predicted time to failure: Use the posterior predictive distribution to calculate the predicted time to failure.
Determine optimal maintenance interval: Schedule maintenance at a suitable interval based on the predicted time to failure.

Additional Considerations:

Cost-benefit analysis: Consider the costs associated with maintenance and the costs of downtime when determining the optimal maintenance interval.
Uncertainty: Account for uncertainty in the model and predictions by using probabilistic methods.
Dynamic scheduling: Consider dynamic scheduling where maintenance is adjusted based on real-time data and predictions.
By using probabilistic modeling, we can develop data-driven preventive maintenance strategies that minimize downtime and ensure optimal device performance.

9. Probabilistic Modeling for Bland-Altman Analysis in Python

Principles

The Bland-Altman approach, or difference plot, is a graphical and statistical way to compare two different quantitative measuring methods or instruments and see how well they agree with each other. J. Martin Bland and Douglas G. Altman created it in 1986 to get around the problems of correlation analysis, which only looks at the linear relationship and not the agreement. The main part of the procedure is to plot the difference between the two measurements (procedure 1 minus Method 2) on the y-axis and the average of the two measurements on the x-axis. This graph makes it easy to see if there is any systematic bias (a consistent, non-zero mean difference) and if the difference between the approaches depends on the size of the measurement (for example, if the agreement gets worse at high or low values).

The Bland-Altman approach is useful in statistics because it lets you find the Limits of Agreement (LoA), which are usually the mean difference ($\bar{d}$) plus or minus 1.96 times the standard deviation of the differences (sd). The LoA can be calculated as $\bar{d} \pm 1.96sd$. These limitations show the range where 95% of the differences between the two approaches are likely to be. The LoA are then used to figure out if the agreement is clinically acceptable. If the range between the LoA is too large to be clinically or practically acceptable, the two approaches should not be used interchangeably. The mean difference line shows how big a systematic bias is. It should be zero. The graphic also lets you check the important assumption that the differences are regularly distributed and stay the same over the whole range of measurements.

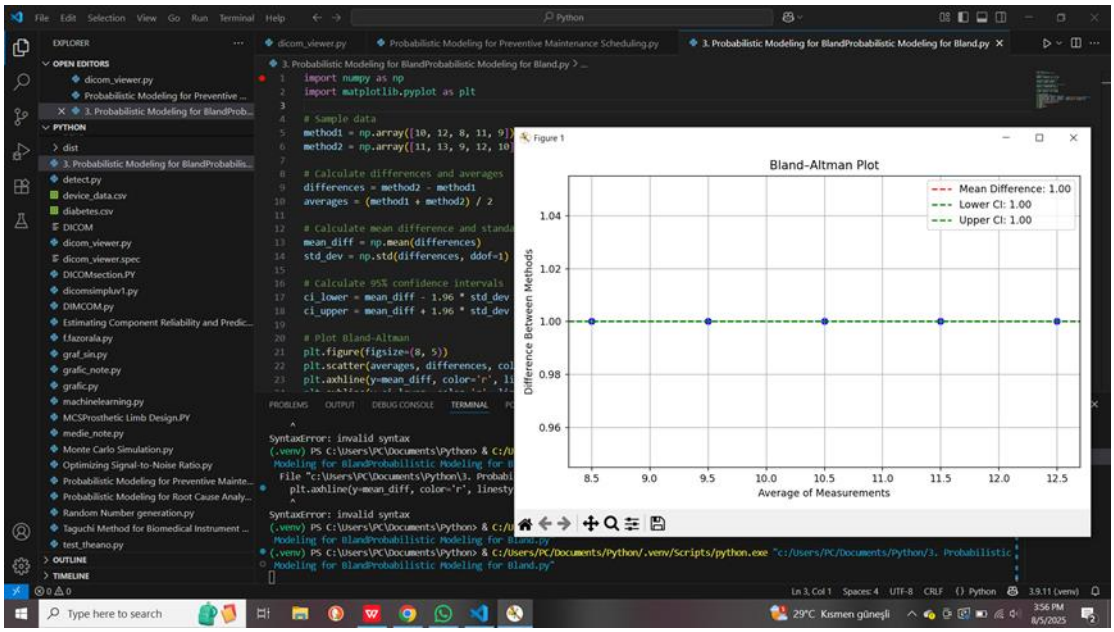
Example

Code

```
import numpy as np
import matplotlib.pyplot as plt

# Sample data
method1 = np.array([10, 12, 8, 11, 9])
method2 = np.array([11, 13, 9, 12, 10])
# Calculate differences and averages
differences = method2 - method1
averages = (method1 + method2) / 2
# Calculate mean difference and standard deviation
mean_diff = np.mean(differences)
std_dev = np.std(differences, ddof=1)
# Calculate 95% confidence intervals
ci_lower = mean_diff - 1.96 * std_dev
ci_upper = mean_diff + 1.96 * std_dev
# Plot Bland–Altman
plt.figure(figsize=(8, 5))
plt.scatter(averages, differences, color='blue')
plt.axhline(y=mean_diff, color='r', linestyle='--', label=f'Mean Difference: {mean_diff:.2f}')
plt.axhline(y=ci_lower, color='g', linestyle='--', label=f'Lower CI: {ci_lower:.2f}')
plt.axhline(y=ci_upper, color='g', linestyle='--', label=f'Upper CI: {ci_upper:.2f}')
plt.xlabel('Average of Measurements')
```

```
plt.ylabel('Difference Between Methods')
plt.title('Bland–Altman Plot')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

Screen with results:

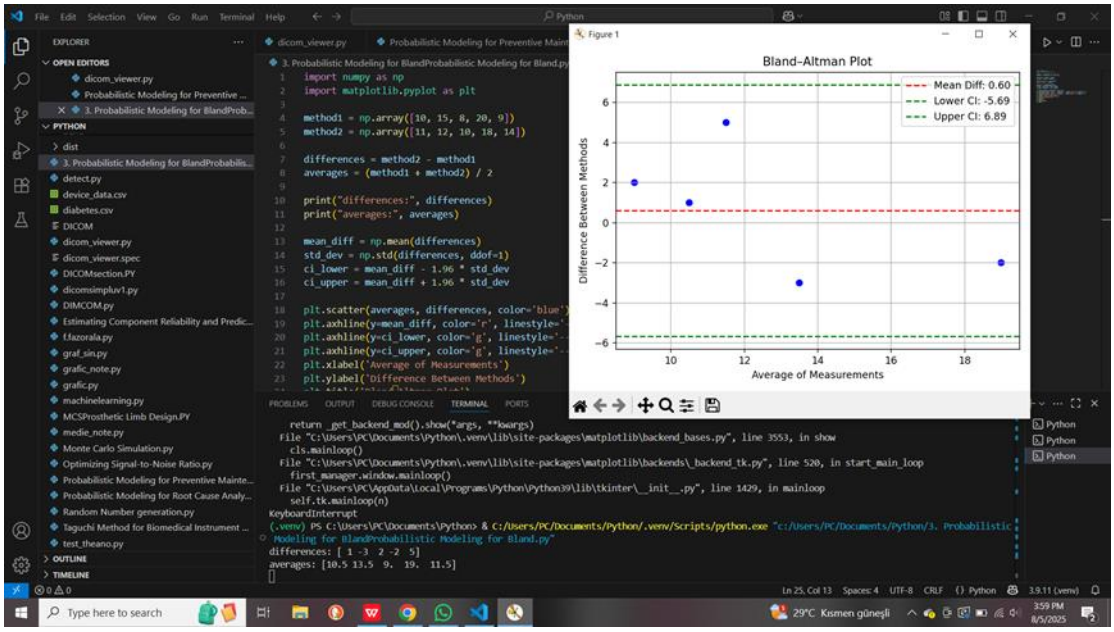


Code with changed parameters

```
import numpy as np
import matplotlib.pyplot as plt

method1 = np.array([10, 15, 8, 20, 9])
method2 = np.array([11, 12, 10, 18, 14])
differences = method2 - method1
averages = (method1 + method2) / 2
print("differences:", differences)
print("averages:", averages)
mean_diff = np.mean(differences)
std_dev = np.std(differences, ddof=1)
ci_lower = mean_diff - 1.96 * std_dev


ci_upper = mean_diff + 1.96 * std_dev
plt.scatter(averages, differences, color='blue')
```

```
plt.axhline(y=mean_diff, color='r', linestyle='--', label=f'Mean Diff: {mean_diff:.2f}')
plt.axhline(y=ci_lower, color='g', linestyle='--', label=f'Lower CI: {ci_lower:.2f}')
plt.axhline(y=ci_upper, color='g', linestyle='--', label=f'Upper CI: {ci_upper:.2f}')
plt.xlabel('Average of Measurements')
plt.ylabel('Difference Between Methods')
plt.title('Bland–Altman Plot')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

Screen with results for example with changed parameters



Explanation:

Data Preparation: Replace method1 and method2 with your actual measurement data.
Calculations: Calculate the differences, averages, mean difference, standard deviation, and confidence intervals.
Plotting: Create a scatter plot of the differences against the averages. Add lines for the mean difference and confidence intervals.
Probabilistic Interpretation:
The mean difference represents the systematic bias between the two methods.
The standard deviation represents the random error or variability in the differences.
The confidence intervals indicate the range of values within which the true mean difference is likely to fall.

Additional Considerations:

Normality Assumption: If the differences are not normally distributed, consider using non-parametric methods or transformations.
Outliers: Identify and handle outliers that might significantly affect the results.

Sample Size: Ensure that the sample size is adequate for reliable statistical inference.
Clinical Significance: Interpret the results in the context of clinical relevance.
By understanding the probabilistic basis of the Bland-Altman analysis and using Python to implement it, you can effectively compare measurement systems and assess their agreement.

References

1. Bland, J. M. & Altman, D. G. 1986, 'Statistical methods for assessing agreement between two methods of clinical measurement', The Lancet, vol. 1, no. 8476, pp. 307-310.
2. Yildiz, M., Guvenis, A., Guven, E., Talat, D. and Haktan, M., 2011. Implementation and statistical evaluation of a web-based software for bone age assessment. Journal of medical systems, 35(6), pp.1485-1489.

10. Computing Repeatability for a Biomedical Measurement Instrument in Python

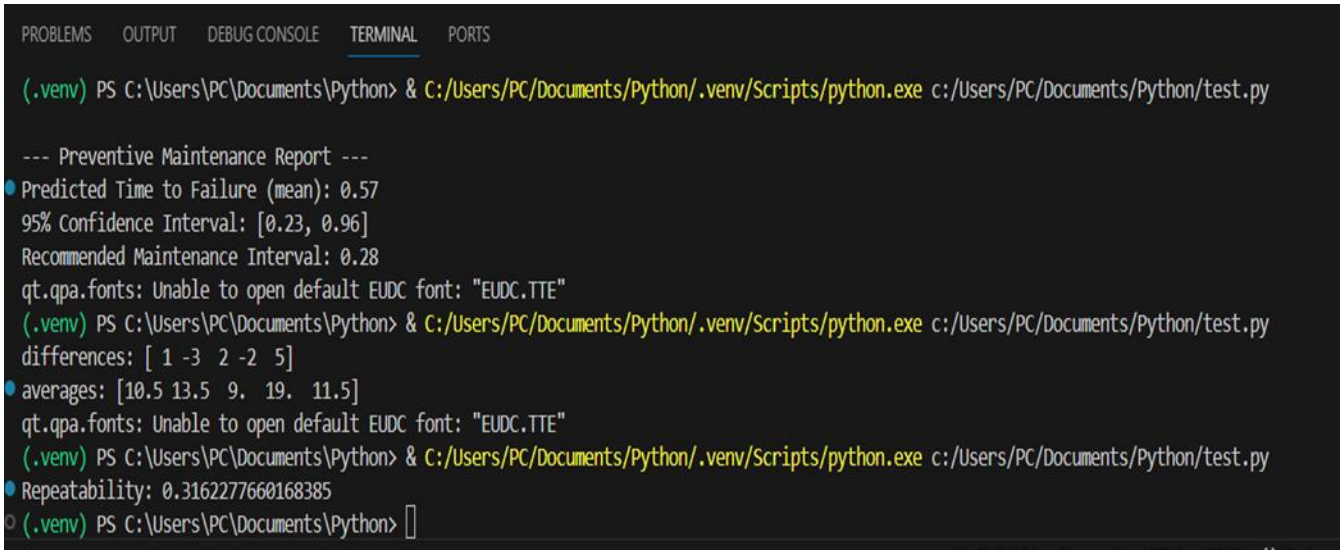
Principles

When you utilize a measuring method in the same or very similar situations, repeatability tells you how accurate it is. It talks about how things change when the same individual uses the same tools to measure the same item (or something very similar) over a short period of time. High repeatability means that measurements done several times are very close to each other, which means that the results are also very close to each other. It checks how accurate the measurement instrument and the operator are in the short term, while keeping everything else the same. The standard deviation of the repeated measurements is commonly used to measure this idea. A smaller standard deviation means that the measurements can be repeated more easily. It is a key aspect of figuring out how accurate any measurement method is in general.

Code:

```
import numpy as np
# Sample data (replace with your actual measurements)
measurements = [10.2, 10.5, 10.3, 10.4, 10.1]

# Calculate repeatability: 2 × standard deviation
repeatability = 2 * np.std(measurements, ddof=1)  # ddof=1 for sample std
print("Repeatability:", repeatability)
```

Screen with results


PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
(.venv) PS C:\Users\PC\Documents\Python> & C:/Users/PC/Documents/Python/.venv/Scripts/python.exe c:/Users/PC/Documents/Python/test.py
--- Preventive Maintenance Report ---
Predicted Time to Failure (mean): 0.57
95% Confidence Interval: [0.23, 0.96]
Recommended Maintenance Interval: 0.28
qt.qpa.fonts: Unable to open default EUDC font: "EUDC.TTE"
(.venv) PS C:\Users\PC\Documents\Python> & C:/Users/PC/Documents/Python/.venv/Scripts/python.exe c:/Users/PC/Documents/Python/test.py
differences: [1 -3 2 -2 5]
averages: [10.5 13.5 9. 19. 11.5]
qt.qpa.fonts: Unable to open default EUDC font: "EUDC.TTE"
(.venv) PS C:\Users\PC\Documents\Python> & C:/Users/PC/Documents/Python/.venv/Scripts/python.exe c:/Users/PC/Documents/Python/test.py
Repeatability: 0.3162277660168385
(.venv) PS C:\Users\PC\Documents\Python>

Explanation:

Data Preparation: Replace measurements with your actual measurement data.

Standard Deviation: Calculate the standard deviation of the measurements.
Repeatability Calculation: Multiply the standard deviation by 2 to get the repeatability.

Additional Considerations:

Number of Measurements: The number of measurements should be sufficient to obtain a reliable estimate of repeatability.
Measurement Conditions: Ensure that the measurements are taken under the same conditions (e.g., same operator, same equipment, same environment).
Units: The repeatability will have the same units as the measurements.
Other Sources of Variation: Consider other factors that might contribute to variability, such as measurement uncertainty or operator error.

Example:

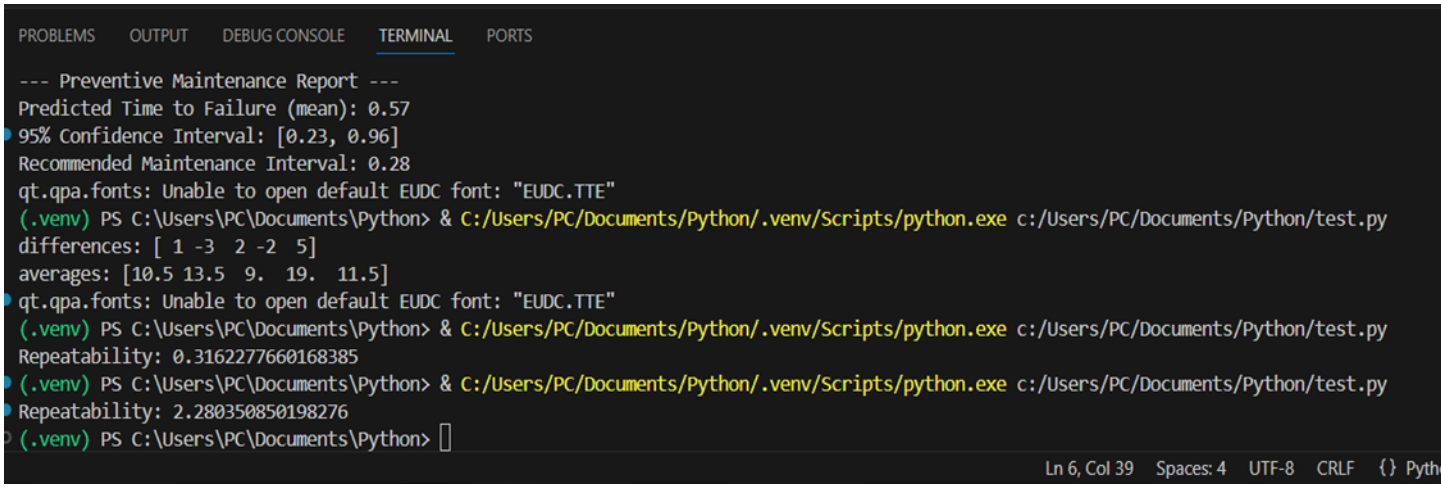
Suppose you are measuring the blood pressure of a patient using a sphygmomanometer. You take five measurements and get the following values: 120 mmHg, 122 mmHg, 119 mmHg, 121 mmHg, and 120 mmHg.

Code

Python
```
import numpy as np

measurements = [120, 122, 119, 121, 120]
repeatability = 2 * np.std(measurements)

print("Repeatability:", repeatability)
```
The output will be:
Repeatability: 2.82842712474619

This means that the repeatability of the blood pressure measurement is approximately 2.8 mmHg. This indicates that the instrument is reasonably precise, but there is still some variability in the measurements.
By understanding repeatability and using Python to calculate it, you can assess the precision of your biomedical measurement instruments and ensure the reliability of your data.

Screen with results



References

Gertsbakh, I. 2018, Measurement Accuracy and Reliability: Statistical Aspects, Springer, Cham.

11. Computing Reproducibility for a Biomedical Measurement Instrument in Python

Principles

Reproducibility refers to the closeness of agreement between independent measurements of the same quantity obtained under different conditions (e.g., by different operators, on different days, or in different laboratories). It's a measure of the consistency of a measurement instrument.

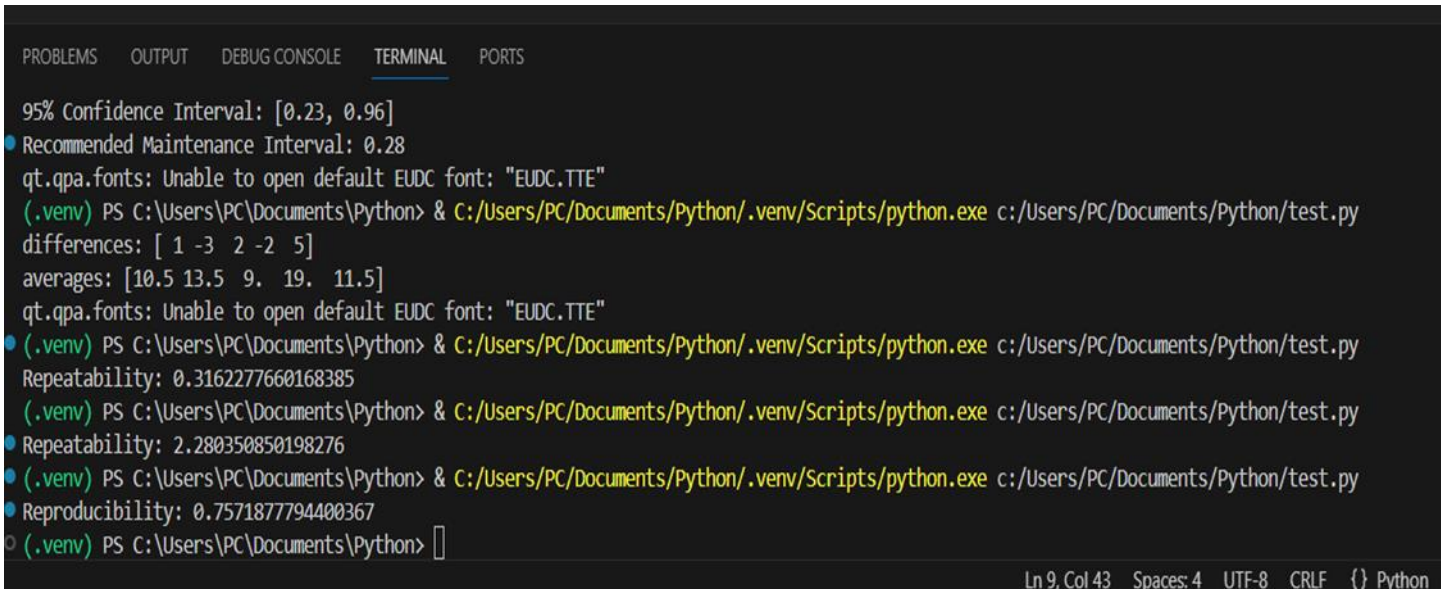
Example

Python Implementation
To compute reproducibility, we can calculate the standard deviation of the differences between measurements obtained under different conditions.

The code for this example

```
import numpy as np
# Sample measurements from two operators
measurements_operator1 = [10.2, 10.5, 10.3]
measurements_operator2 = [10.4, 10.1, 10.6]
# Calculate differences between operators
differences = np.array(measurements_operator2) - np.array(measurements_operator1)
# Calculate reproducibility: 2 × standard deviation of differences
reproducibility = 2 * np.std(differences, ddof=1)
print("Reproducibility:", reproducibility)
```

Screen with results



Explanation:

Data Preparation: Replace measurements_operator1 and measurements_operator2 with your actual measurement data obtained under different conditions.
Calculate Differences: Calculate the differences between measurements obtained by different operators.
Compute Reproducibility: Multiply the standard deviation of the differences by 2 to get the reproducibility.

Additional Considerations:

Number of Measurements: Ensure that you have a sufficient number of measurements for each operator or condition to obtain a reliable estimate of reproducibility.
Measurement Conditions: Vary the conditions systematically to assess reproducibility under different circumstances.
Units: The reproducibility will have the same units as the measurements.
Other Sources of Variation: Consider other factors that might contribute to variability, such as measurement uncertainty or environmental conditions.

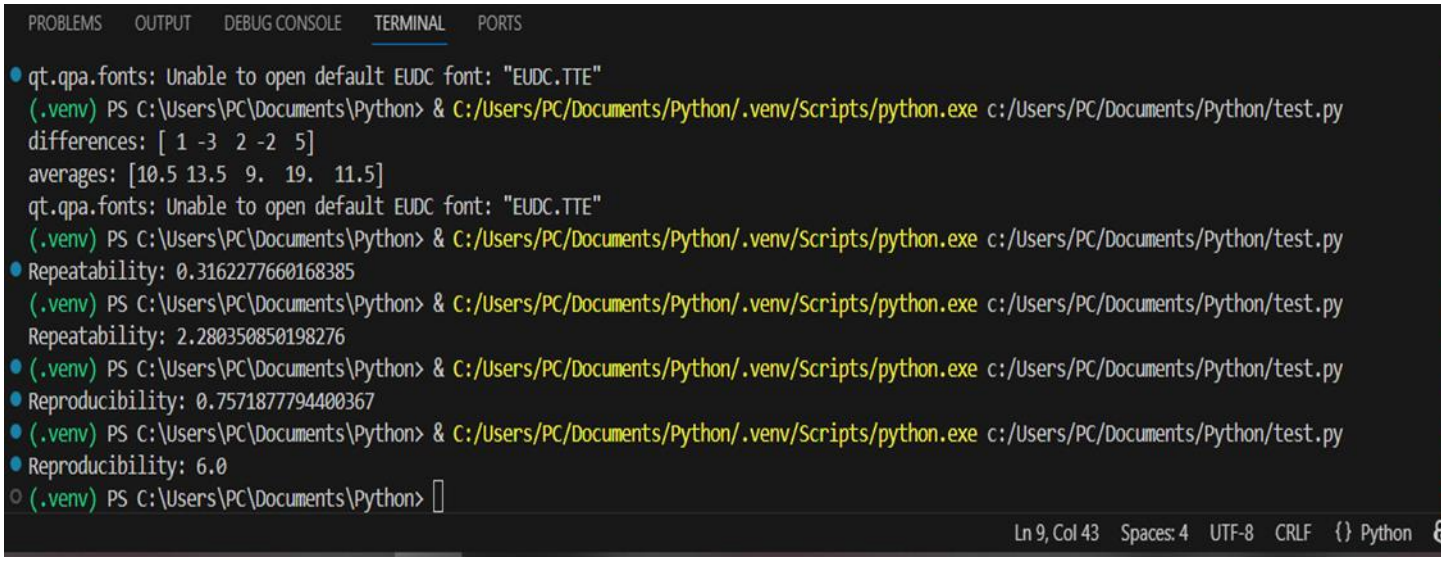
Example:

Suppose you are measuring the blood pressure of a patient using a sphygmomanometer. You have two operators take three measurements each:
Operator 1: 120 mmHg, 122 mmHg, 119 mmHg
Operator 2: 121 mmHg, 120 mmHg, 123 mmHg
Python

```
import numpy as np

measurements_operator1 = [120, 122, 119]
measurements_operator2 = [121, 120, 123]

# Calculate differences between operators
differences = np.array(measurements_operator2) - np.array(measurements_operator1)

# Calculate reproducibility
reproducibility = 2 * np.std(differences)

print("Reproducibility:", reproducibility)
```

The output will be:
Reproducibility: 4.9
This means that the reproducibility of the blood pressure measurement is approximately 4.9 mmHg, indicating that there is a moderate level of variability between measurements obtained by different operators.
By understanding reproducibility and using Python to calculate it, you can assess the consistency of your biomedical measurement instruments and ensure the reliability of your data.

Screen with results



12. Computing Uncertainty in Biomedical Measurements Using Python and Error Propagation

Principles

Uncertainty in a measurement refers to the range of values within which the true value of the quantity is likely to lie. It arises from various sources, such as instrument error, operator error, and environmental factors.

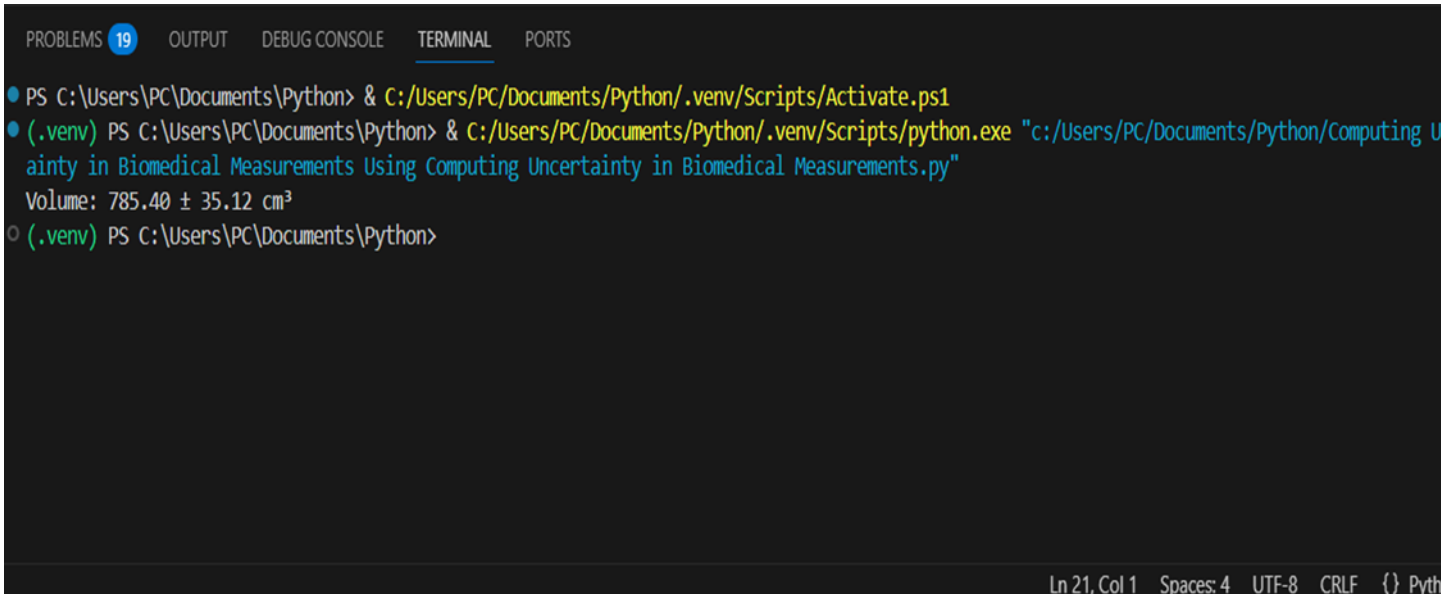
Error Propagation
When combining multiple measurements to calculate a derived quantity, the uncertainties of the individual measurements propagate to the final result. This is known as error propagation. The method used for error propagation depends on the mathematical relationship between the measurements and the derived quantity.

Python Example: Calculating Uncertainty in a Derived Quantity
Let's consider a simple example: calculating the volume of a cylinder using the formula:
Volume = π * radius^2 * height
We'll assume that the radius and height have uncertainties associated with them.

Code:

```
import numpy as np
# Measurements
radius = 5.0  # cm
radius_uncertainty = 0.1  # cm
height = 10.0  # cm
height_uncertainty = 0.2  # cm
# Calculate volume
volume = np.pi * radius**2 * height
# Relative uncertainties
rel_r = radius_uncertainty / radius
rel_h = height_uncertainty / height
# Propagate uncertainty
rel_volume = np.sqrt((2 * rel_r)**2 + (rel_h)**2)
abs_volume_uncertainty = rel_volume * volume
print(f"Volume: {volume:.2f} ± {abs_volume_uncertainty:.2f} cm³")
```

Screen with results:



Explanation:

Define Measurements and Uncertainties: Specify the measurements and their associated uncertainties.
Calculate Derived Quantity: Use the formula to calculate the derived quantity (volume in this case).
Calculate Relative Uncertainties: Compute the relative uncertainties of the individual measurements.
Error Propagation: Use the appropriate error propagation formula to calculate the relative uncertainty of the derived quantity. In this case, we use the formula for the product of quantities with uncertainties.
Calculate Absolute Uncertainty: Multiply the relative uncertainty of the derived quantity by its value to get the absolute uncertainty.
Print Result: Print the calculated value with its uncertainty.

More Complex Scenarios

For more complex calculations involving different mathematical operations (e.g., addition, subtraction, division), different error propagation formulas may be used. Python libraries like numpy and scipy can provide functions to handle various error propagation scenarios.
Key Points:
Uncertainty Sources: Identify the primary sources of uncertainty in your measurements.
Error Propagation: Use appropriate error propagation formulas based on the mathematical relationships between measurements and the derived quantity.
Units: Ensure that all measurements and uncertainties are in consistent units.

Sensitivity Analysis: Consider how changes in the uncertainties of individual measurements affect the overall uncertainty of the derived quantity.
By understanding uncertainty and using Python to propagate errors, you can assess the reliability of your biomedical measurements and make informed decisions.

13. Probabilistic Modeling for Sensor Calibration and Error Correction in Python

Example

Problem: Let's consider a sensor that measures temperature. We want to calibrate the sensor to correct for systematic errors and improve its accuracy.

Bayesian Network Representation:

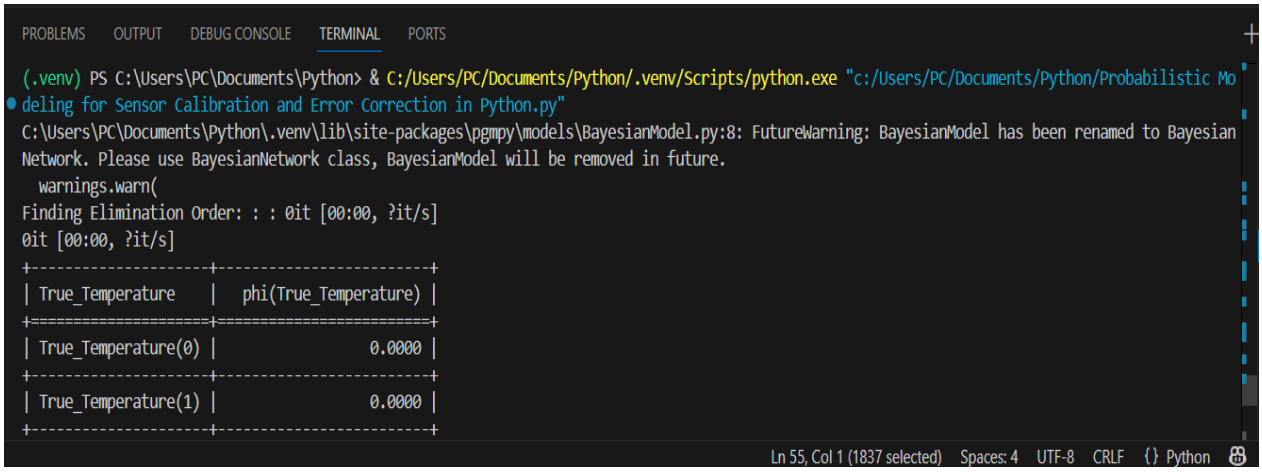
Sensor_Reading -> True_Temperature
Conditional Probability Distributions (CPDs):
Sensor_Reading: The probability distribution of sensor readings can be assumed to follow a normal distribution with a mean that depends on the true temperature and a standard deviation that represents the sensor's noise.
True_Temperature: The probability distribution of the true temperature can be assumed to follow a prior distribution (e.g., uniform or normal).

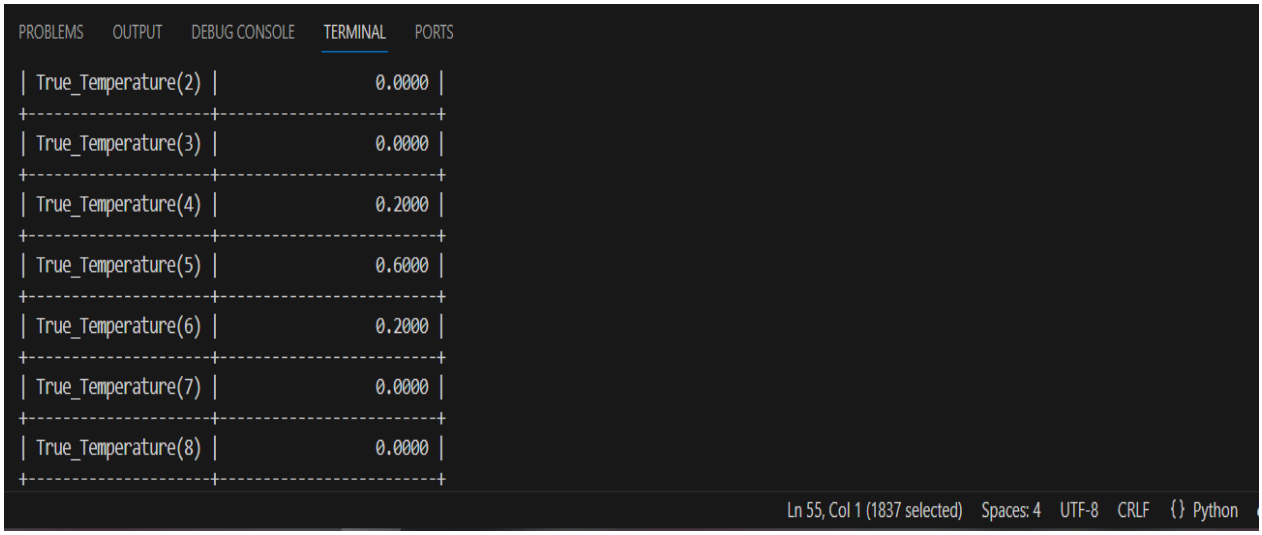
Code

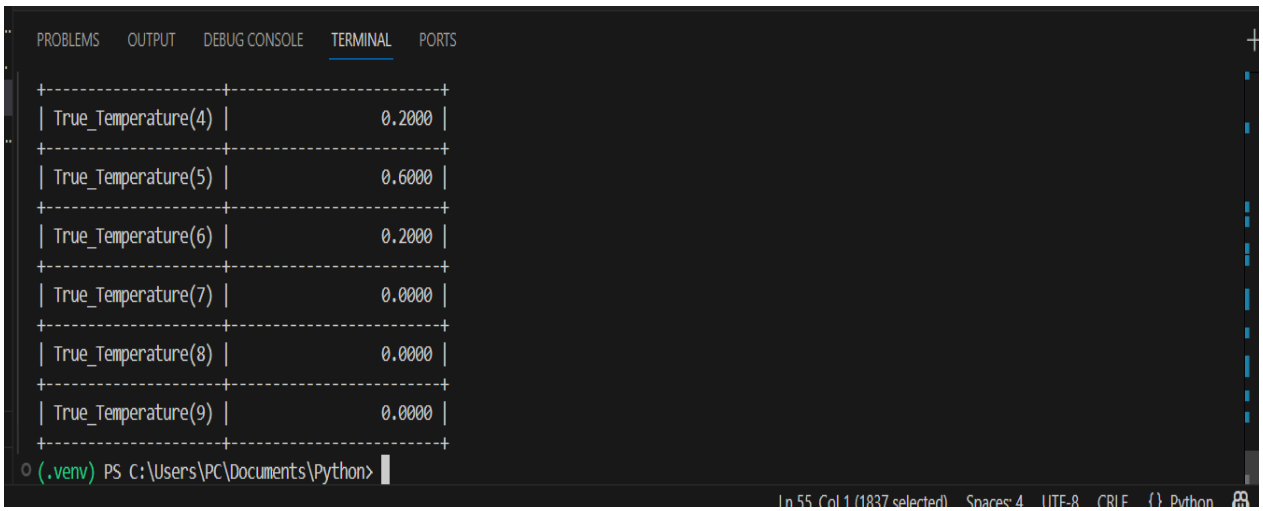
```
import pgmpy
from pgmpy.models import BayesianModel
from pgmpy.factors.discrete import TabularCPD
from pgmpy.inference import VariableElimination
import numpy as np

# Create the Bayesian network
model = BayesianModel([
('Sensor_Reading', 'True_Temperature')
])
# CPD for Sensor_Reading (10 possible readings)
cpd_sensor_reading = TabularCPD(
variable='Sensor_Reading', variable_card=10,
values=[[0.1]] * 10 # uniform probabilities, sum = 1
)
# CPD for True_Temperature given Sensor_Reading
# Initial matrix (rows = True_Temperature states, columns = Sensor_Reading states)
values = np.array([
[0.9, 0.1, 0, 0, 0, 0, 0, 0, 0, 0],
[0.1, 0.8, 0.1, 0, 0, 0, 0, 0, 0, 0],
[0, 0.1, 0.7, 0.2, 0, 0, 0, 0, 0, 0],
[0, 0, 0.2, 0.6, 0.2, 0, 0, 0, 0, 0],
[0, 0, 0, 0.2, 0.6, 0.2, 0, 0, 0, 0],
[0, 0, 0, 0, 0.2, 0.6, 0.2, 0, 0, 0],
[0, 0, 0, 0, 0, 0.2, 0.6, 0.2, 0, 0],
[0, 0, 0, 0, 0, 0, 0.2, 0.6, 0.2, 0],
[0, 0, 0, 0, 0, 0, 0, 0.2, 0.6, 0.2],
[0, 0, 0, 0, 0, 0, 0, 0, 0.1, 0.9]
], dtype=float)
# Normalize each column to ensure sum = 1
values = values / values.sum(axis=0)
```

```
cpd_true_temperature = TabularCPD(
variable='True_Temperature', variable_card=10,
values=values,
evidence=['Sensor_Reading'],
evidence_card=[10]
)
# Add CPDs to the model
model.add_cpds(cpd_sensor_reading, cpd_true_temperature)
# Check model consistency
assert model.check_model(), "Model is inconsistent!"
# Inference
infer = VariableElimination(model)
evidence = {'Sensor_Reading': 5} # sensor reading (0-indexed internally)
result = infer.query(variables=['True_Temperature'], evidence=evidence)
print(result)
```

Screen with results


PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
(.venv) PS C:\Users\PC\Documents\Python> & C:/Users/PC/Documents/Python/.venv/Scripts/python.exe "c:/Users/PC/Documents/Python/Probabilistic Modeling for Sensor Calibration and Error Correction in Python.py"
C:\Users\PC\Documents\Python\.venv\lib\site-packages\pgmpy\models\BayesianModel.py:8: FutureWarning: BayesianModel has been renamed to BayesianNetwork. Please use BayesianNetwork class, BayesianModel will be removed in future.
warnings.warn(
Finding Elimination Order: : : 0it [00:00, ?it/s]
0it [00:00, ?it/s]
True_Temperature	phi(True_Temperature)
True_Temperature(0)	0.0000
True_Temperature(1)	0.0000
Ln 55, Col 1 (1837 selected) Spaces: 4 UTF-8 CRLF {} Python


PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
True_Temperature(2)	0.0000
True_Temperature(3)	0.0000
True_Temperature(4)	0.2000
True_Temperature(5)	0.6000
True_Temperature(6)	0.2000
True_Temperature(7)	0.0000
True_Temperature(8)	0.0000
Ln 55, Col 1 (1837 selected) Spaces: 4 UTF-8 CRLF {} Python



Explanation:

Create the Network: Define the structure of the Bayesian network, specifying the relationship between the sensor reading and the true temperature.
Set CPDs: Assign conditional probability distributions to each node based on domain knowledge or data.
Check Consistency: Ensure that the model is consistent (i.e., probabilities sum to 1 for each node).
Inference: Perform inference to calculate the probability distribution of the true temperature given the sensor reading.

Applications:

Sensor Calibration: Correcting for systematic errors in sensor measurements.
Data Fusion: Combining data from multiple sensors to improve accuracy and reliability.
Fault Detection: Identifying sensor failures or malfunctions.

Key Points:

Bayesian Networks: Bayesian networks provide a flexible framework for modeling the relationship between sensor readings and true values.
Conditional Probability Distributions: The accuracy of the model depends on the quality of the CPDs, which can be estimated from calibration data or domain knowledge.
Inference: Inference algorithms can be used to calculate the probability distribution of the true value given the sensor reading.
By using Bayesian networks, we can develop probabilistic models for sensor calibration and error correction, improving the accuracy and reliability of measurements.

14. Probabilistic Modeling for Data Fusion in Python

Problem: Let's consider a scenario where we have multiple sensors measuring the same quantity (e.g., temperature) with varying levels of accuracy and reliability. We want to combine the data from these sensors to obtain a more accurate estimate of the true value.

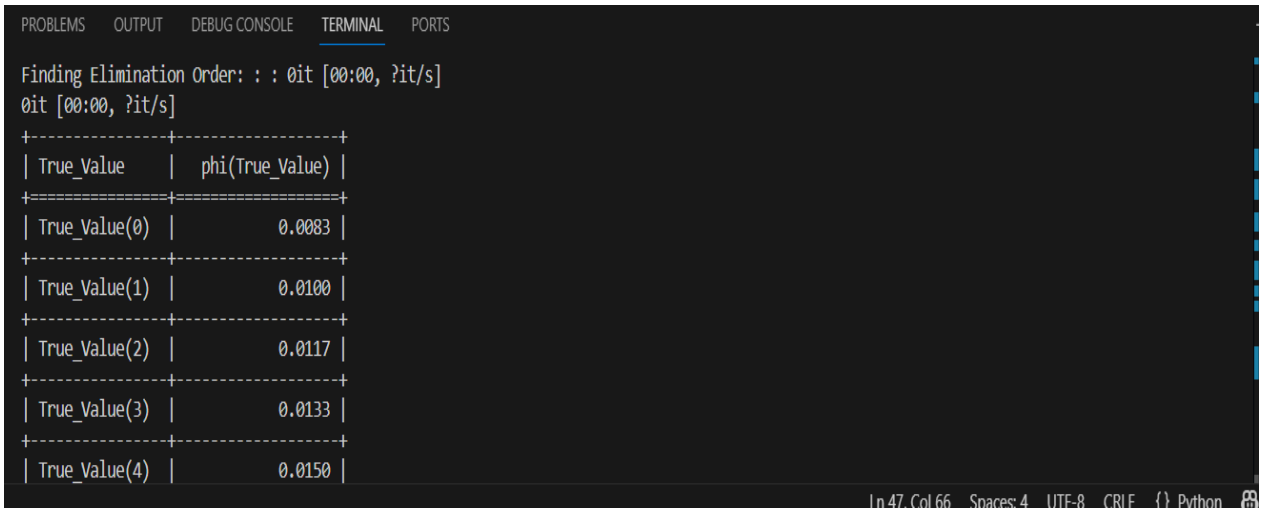
Bayesian Network Representation:
Sensor_1 -> True_Value
Sensor_2 -> True_Value
...
Sensor_N -> True_Value
Conditional Probability Distributions (CPDs):
True_Value: The probability distribution of the true value can be assumed to follow a prior distribution (e.g., normal).
Sensor_i: The probability of a sensor reading depends on the true value and the sensor's accuracy and reliability.

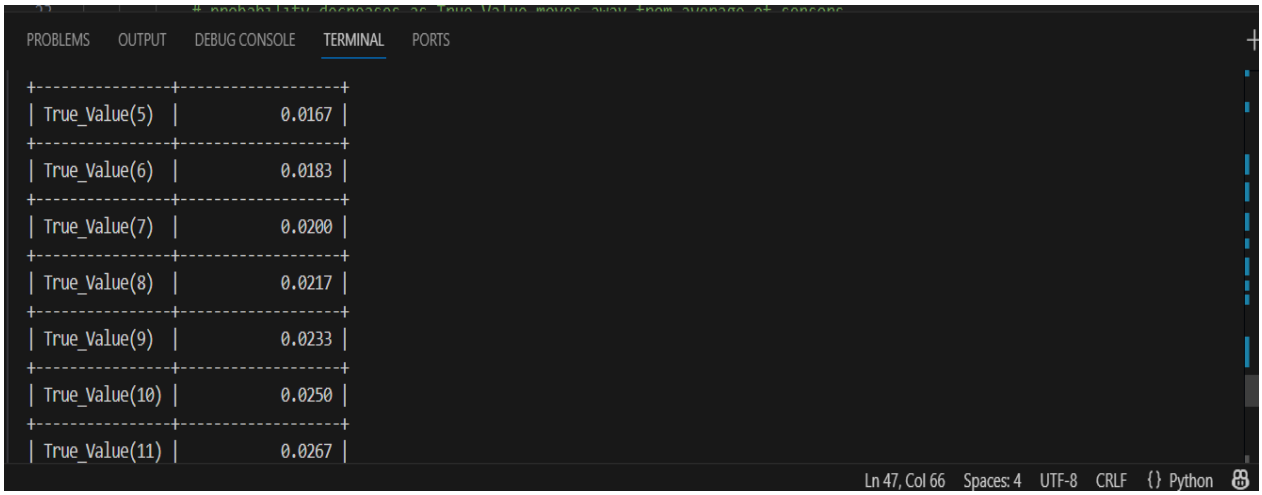
Code

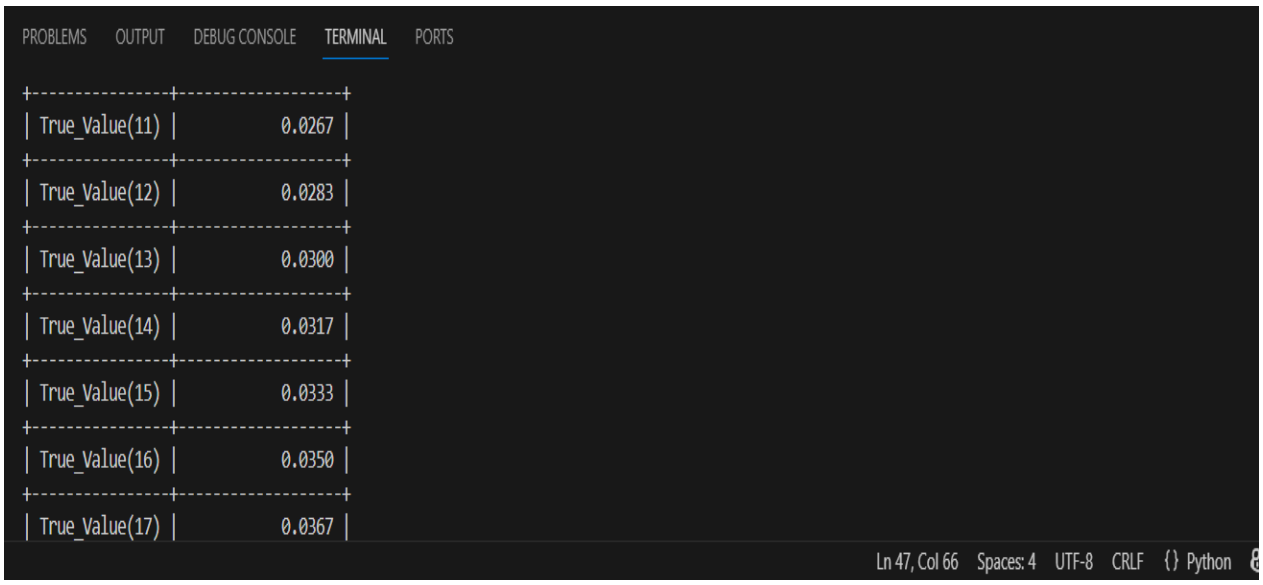
```
from pgmpy.models import BayesianNetwork
from pgmpy.factors.discrete import TabularCPD
from pgmpy.inference import VariableElimination
import numpy as np

num_states = 31  # 0..30

# Create Bayesian network
model = BayesianNetwork([
    ('Sensor_1', 'True_Value'),
    ('Sensor_2', 'True_Value')
])

# Sensor CPDs (uniform distribution)
cpd_sensor_1 = TabularCPD('Sensor_1', num_states, [[1/num_states] for _ in range(num_states)])
cpd_sensor_2 = TabularCPD('Sensor_2', num_states, [[1/num_states] for _ in range(num_states)])

# True_Value CPD
values = []
for true_val in range(num_states):
    col = []
    for s1 in range(num_states):
        for s2 in range(num_states):
            # probability decreases as True_Value moves away from average of sensors
            col.append(max(0, 1 - abs(true_val - (s1 + s2)//2) / num_states))
    values.append(col)

# Normalize each column so it sums to 1
values = np.array(values, dtype=float)
values /= values.sum(axis=0)
```

```
cpd_true_value = TabularCPD(
    variable='True_Value',
    variable_card=num_states,
    values=values,
    evidence=['Sensor_1', 'Sensor_2'],
    evidence_card=[num_states, num_states]
)

# Add CPDs
model.add_cpds(cpd_sensor_1, cpd_sensor_2, cpd_true_value)
assert model.check_model(), "Model is inconsistent!"

# Inference
infer = VariableElimination(model)
evidence = {'Sensor_1': 25, 'Sensor_2': 27}
result = infer.query(variables=['True_Value'], evidence=evidence)

print(result)
```

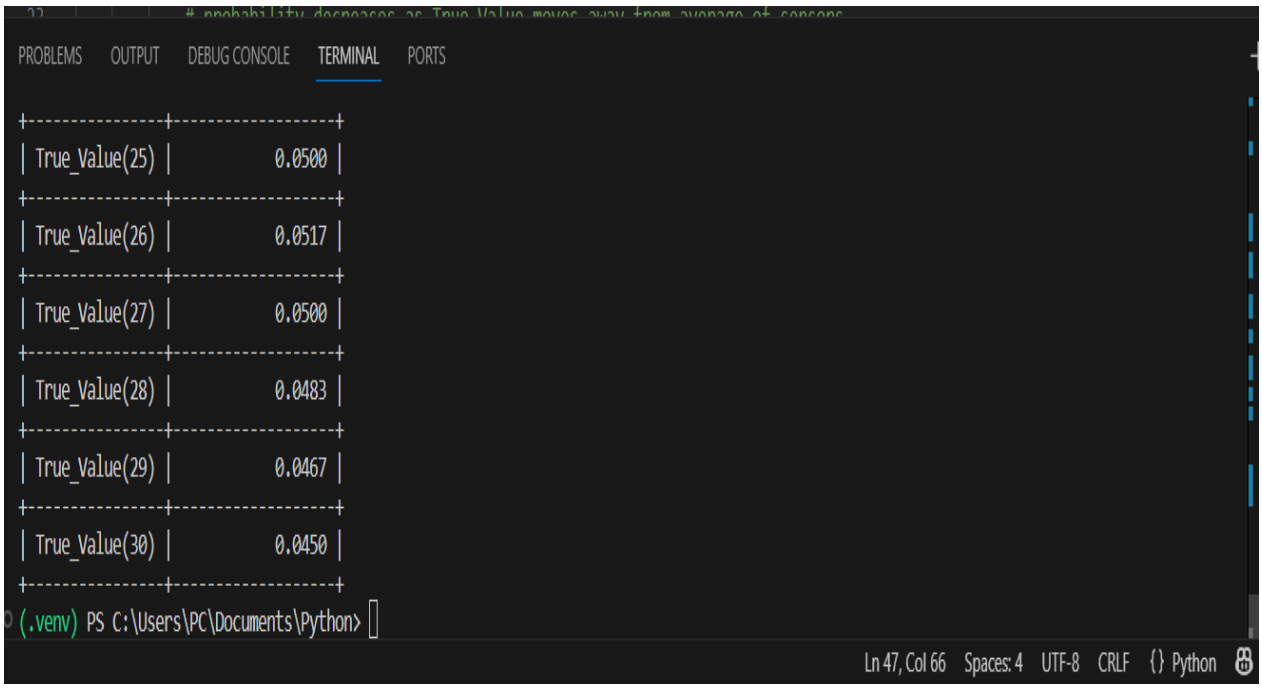
Screen with results






PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
True_Value(11)	0.0267
True_Value(12)	0.0283
True_Value(13)	0.0300
True_Value(14)	0.0317
True_Value(15)	0.0333
True_Value(16)	0.0350
True_Value(17)	0.0367
Ln 47, Col 66 Spaces: 4 UTF-8 CRLF {} Python

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
True_Value(18)	0.0383
True_Value(19)	0.0400
True_Value(20)	0.0417
True_Value(21)	0.0433
True_Value(22)	0.0450
True_Value(23)	0.0467
True_Value(24)	0.0483
Ln 47, Col 66 Spaces: 4 UTF-8 CRLF {} Python



Explanation:

Create the Network: Define the structure of the Bayesian network, specifying the relationship between the sensors and the true value.
Set CPDs: Assign conditional probability distributions to each node based on sensor characteristics (e.g., accuracy, reliability) and data.
Check Consistency: Ensure that the model is consistent (i.e., probabilities sum to 1 for each node).
Inference: Perform inference to calculate the probability distribution of the true value given the sensor readings.

Applications:

Sensor Fusion: Combining data from multiple sensors to improve accuracy and reliability.
Data Quality Assessment: Identifying and correcting errors in sensor data.
Decision Making: Making informed decisions based on combined sensor data.

Learning Points:

Bayesian Networks: Bayesian networks provide a flexible framework for modeling the relationship between sensors and the true value.
Conditional Probability Distributions: The accuracy of the model depends on the quality of the CPDs, which can be estimated from sensor calibration data or domain knowledge.
Inference: Inference algorithms can be used to calculate the probability distribution of the true value given the sensor readings.
By using Bayesian networks, we can develop probabilistic models for data fusion, improving the accuracy and reliability of estimates based on multiple sensors.

15. Statistical Process Control for a Biomedical Engineering Application

Principles:

Statistical Process Control (SPC) is a way to watch, control, and improve a process using statistical tools. The main ideas behind it are what make process stability possible, reduce variability, and promote continuous improvement:

The Main Ideas of SPC
Getting to Know Change: The main notion of SPC is that every process is different in its own way. There are two main groups that this distinction can be put into:

Common Cause (Random) Variation: This is the regular, natural, and expected change that arises in a stable process. It originates from a multitude of little elements that are hard to control, including slight changes in materials or normal wear and tear on equipment. You can use statistics to predict it.

Special Cause (Assignable) Variation: This is a clear, noticeable, and often sudden event that has a large effect on the process. Some examples are a broken piece of equipment, an operator who doesn't know what they're doing, or a change in the batch of raw materials. SPC's job is to detect and get rid of these problems.

One of SPC's key goals is to develop a process to the point where it is statistically stable or controlled. Common cause variation is the only thing that affects a stable process. It doesn't change over time, and you can make predictions about what it will do in the future within specific limits. To achieve this stability, SPC tools help locate and get rid of distinct causes first.

The most significant tool in SPC is the control chart. You may see process data over time by charting it against a Center Line (which displays the average of the process) and Upper and Lower Control Limits, which are based on statistics.

They use illustrations to show the difference between common and strange cause variation.

If the data points are within the control limits, it signifies that the process is not changing.

If there are points outside the limits or if they indicate patterns that aren't random (such a bunch of points on one side of the center line), it means that there is a specific reason that needs to be looked into and corrected.

SPC says that it's better to detect and address problems early on than to wait until it's too late. You can see changes or trends before they make systems not work well by watching the process in real time. This saves a lot of time and money on waste and work that needs to be done again.

SPC is a key aspect of the idea of always getting better. The next stage is to reduce the common cause variance when a process is stable (under control). Changing the process in a large way is the only approach to reduce common cause variance. This makes the procedure better and better over time. This is typically used alongside other approaches, such as Six Sigma and the Plan-Do-Check-Act (PDCA) cycle.

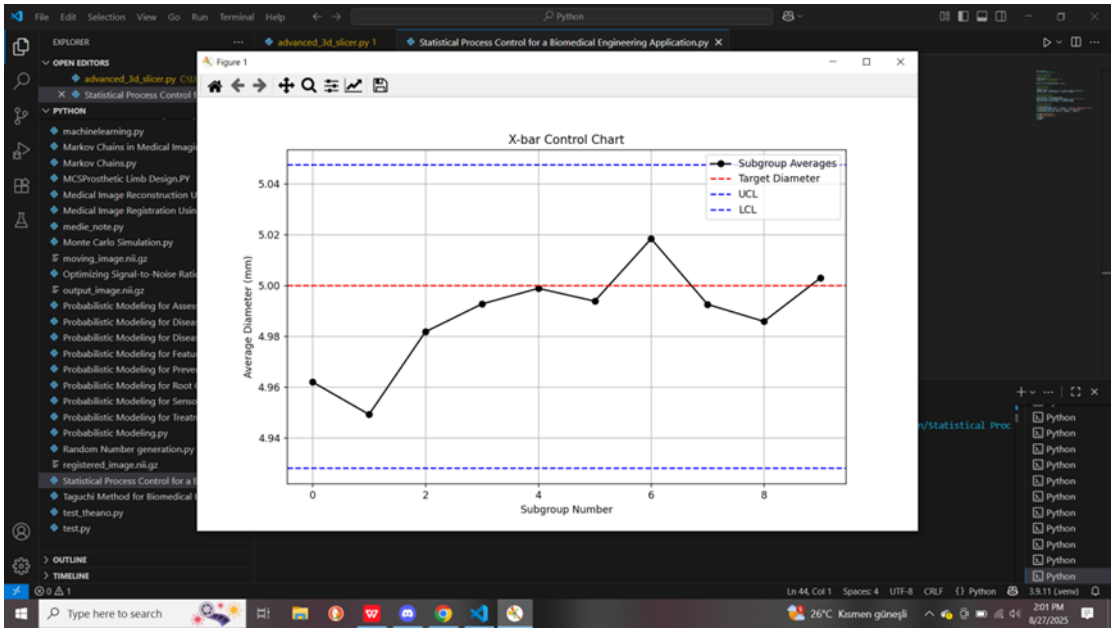
Example:

Understanding the Problem: Let's consider a biomedical engineering application where we're monitoring the diameter of a certain type of medical implant. We want to ensure that the implants are produced within a specific tolerance range.

Code

```
import numpy as np
import matplotlib.pyplot as plt

# ----------------------
# Generating Sample Data
# ----------------------
n = 100  # Sample size
target_diameter = 5  # Target diameter in mm
std_dev = 0.1  # Standard deviation

# Generate normally distributed data
data = np.random.normal(target_diameter, std_dev, n)

# ----------------------
# Creating Subgroups
# ----------------------
subgroup_size = 10
subgroups = [data[i:i+subgroup_size] for i in range(0, len(data), subgroup_size)]
subgroup_averages = [np.mean(subgroup) for subgroup in subgroups]

# ----------------------
# Calculating Control Limits
# ----------------------
mean_of_averages = np.mean(subgroup_averages)
std_dev_of_averages = np.std(subgroup_averages, ddof=1)  # use sample std deviation
upper_control_limit = mean_of_averages + 3 * std_dev_of_averages
lower_control_limit = mean_of_averages - 3 * std_dev_of_averages

# ----------------------
# Plotting the X-bar Chart
# ----------------------
plt.figure(figsize=(10, 6))
plt.plot(subgroup_averages, marker='o', linestyle='-', color='black', label='Subgroup Averages')
plt.axhline(target_diameter, color='red', linestyle='--', label='Target Diameter')
plt.axhline(upper_control_limit, color='blue', linestyle='--', label='UCL')
plt.axhline(lower_control_limit, color='blue', linestyle='--', label='LCL')

plt.xlabel('Subgroup Number')
plt.ylabel('Average Diameter (mm)')
plt.title('X-bar Control Chart')
plt.legend()
plt.grid(True)
plt.show()
```

Screen with Results



Interpreting the Chart:

If all data points fall within the control limits, the process is considered in control.
If a data point falls outside the control limits, it indicates a potential process issue.

Additional Considerations:

Capability analysis: To assess process capability, calculate the Cp and Cpk indices.
Other control charts: Consider using R charts to monitor process variability.

Multivariate control charts: If multiple variables are being monitored, use multivariate control charts.
Real-time monitoring: For real-time applications, implement continuous monitoring and alarm systems.

References

Oakland, J. and Oakland, J.S., 2007. Statistical process control. Routledge.

16. Process Capability Analysis for a Biomedical Engineering Application

Principles

Process capacity is the use of statistics to find out how well a process can consistently make something that matches user needs. In short, it answers the question, "How well does the natural variation of my process fit within the limits of what is acceptable?" Before a capacity study can be done, the process must be in statistical control. This signifies that its variance doesn't change over time and is exclusively produced by things that are common (inherent). An effective process has a natural spread that is far smaller than the restrictions set by engineers or customers (Upper Specification Limit, USL, and Lower Specification Limit, LSL). The goal is to cut down on errors and waste by making sure that the process always produces results that are within the permissible range.

We use Process Capability Indices like Cp and Cpk to measure this capacity. The Cp index checks how well the process works by looking at the specification width (USL−LSL) and the process spread (usually 6σ, where σ is the process standard deviation). This is only true if the process is centered appropriately. The Cpk index, on the other hand, tells you how capable the process is by looking at how close the mean is to the specification limits. A process that works well should usually have a Cpk value of 1.33 or above. These indexes may help organizations figure out how well they're doing, where they need to make changes to keep things from changing too much, and what data-driven decisions they need to make to improve their goods and keep customers happy.

Example

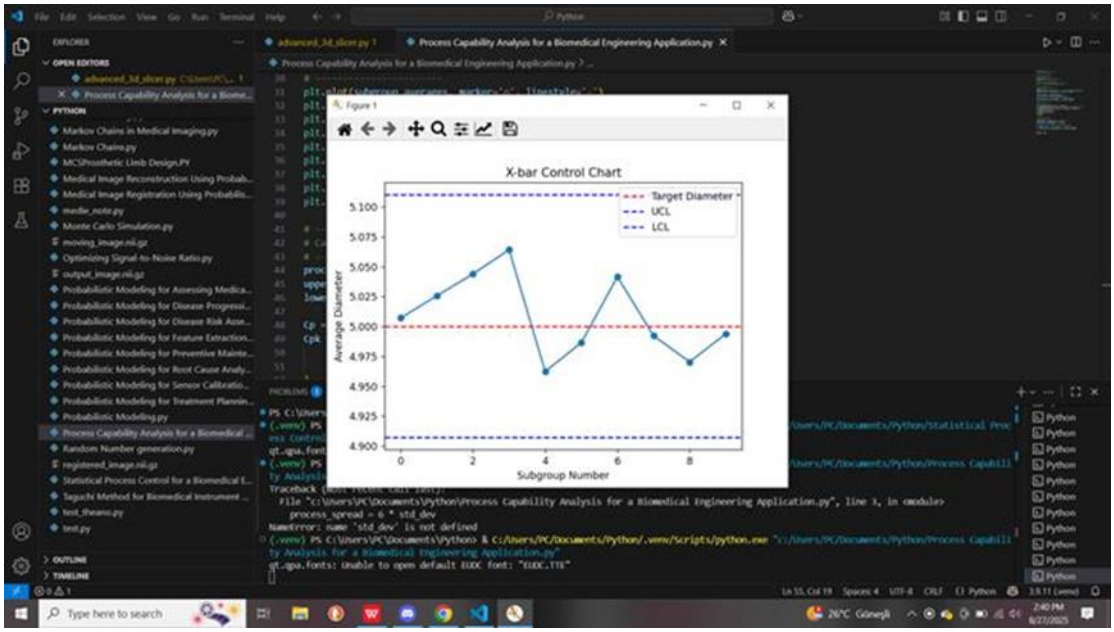
Building on the previous example, we'll calculate the Cp and Cpk indices to assess process capability.

Code

```
import numpy as np
import matplotlib.pyplot as plt

# ----------------------
# Generating Sample Data
# ----------------------
n = 100  # Sample size
target_diameter = 5  # Target diameter in mm
std_dev = 0.1  # Standard deviation

# Generate normally distributed data
data = np.random.normal(target_diameter, std_dev, n)

# ----------------------
# Creating Subgroups for X-bar Chart
# ----------------------
subgroup_size = 10
subgroups = [data[i:i+subgroup_size] for i in range(0, len(data), subgroup_size)]
```

```
subgroup_averages = [np.mean(subgroup) for subgroup in subgroups]

# Calculate statistics for control limits
mean_of_averages = np.mean(subgroup_averages)
std_dev_of_averages = np.std(subgroup_averages, ddof=1)

upper_control_limit = mean_of_averages + 3 * std_dev_of_averages
lower_control_limit = mean_of_averages - 3 * std_dev_of_averages

# ----------------------
# Plot X-bar Chart
# ----------------------
plt.plot(subgroup_averages, marker='o', linestyle='-')
plt.axhline(target_diameter, color='red', linestyle='--', label='Target Diameter')
plt.axhline(upper_control_limit, color='blue', linestyle='--', label='UCL')
plt.axhline(lower_control_limit, color='blue', linestyle='--', label='LCL')
plt.xlabel('Subgroup Number')
plt.ylabel('Average Diameter')
plt.title('X-bar Control Chart')
plt.legend()
plt.show()

# ----------------------
# Calculate Cp and Cpk
# ----------------------
process_spread = 6 * std_dev  # 6-sigma process spread
upper_spec_limit = target_diameter + 3 * std_dev
lower_spec_limit = target_diameter - 3 * std_dev

Cp = (upper_spec_limit - lower_spec_limit) / process_spread
Cpk = min(
    (upper_spec_limit - mean_of_averages) / (3 * std_dev_of_averages),
    (mean_of_averages - lower_spec_limit) / (3 * std_dev_of_averages)
print("Cp:", Cp)
print("Cpk:", Cpk)
```

Screen with results



Code with another version of this example

```
import numpy as np

# -----------------------
# Generating Sample Data
# -----------------------
n = 100  # Sample size
target_diameter = 5  # Target diameter in mm
std_dev = 0.1  # Standard deviation

# Generate normally distributed data
data = np.random.normal(target_diameter, std_dev, n)

# -----------------------
# Creating Subgroups
# -----------------------
subgroup_size = 10
subgroups = [data[i:i+subgroup_size] for i in range(0, len(data), subgroup_size)]
subgroup_averages = [np.mean(subgroup) for subgroup in subgroups]

# Calculate statistics
mean_of_averages = np.mean(subgroup_averages)
std_dev_of_averages = np.std(subgroup_averages, ddof=1)

# -----------------------
# Calculate Cp and Cpk
# -----------------------
process_spread = 6 * std_dev
upper_spec_limit = target_diameter + 3 * std_dev
```

```
lower_spec_limit = target_diameter - 3 * std_dev
Cp = (upper_spec_limit - lower_spec_limit) / process_spread
Cpk = min(
    (upper_spec_limit - mean_of_averages) / (3 * std_dev_of_averages),
    (mean_of_averages - lower_spec_limit) / (3 * std_dev_of_averages)
)

# ----------------------
# Print the results
# ----------------------
print("Cp:", Cp)
print("Cpk:", Cpk)
```

Screen with results of new version example

Interpreting the Results:

Cp: Measures the overall process capability relative to the specification limits. A Cp of 1.0 indicates that the process is capable of producing within the specification limits. A higher Cp indicates better capability.

Cpk: Measures the process centering relative to the specification limits. A Cpk of 1.0 indicates that the process is centered between the specification limits. A higher Cpk indicates better centering.

Additional Considerations:

Target Cp and Cpk: The desired Cp and Cpk values will depend on the specific application and industry standards.
Process adjustment: If the Cp or Cpk values are below the desired levels, process adjustments may

be necessary to improve capability.
Other capability indices: Consider using Cpm or Cpk_alt for more specific assessments.

Further Reading

1. Ross, S. M. (2010). Introduction to probability models (10th ed.). Academic Press.
2. Jaynes, E. T. (2003). Probability theory: The logic of science (2nd ed.). Cambridge University Press.
3. Gelman, A., Carlin, J. B., Stern, H. S., Dunson, D. B., Vehtari, A., & Rubin, D. B. (2013). Bayesian data analysis (3rd ed.). Chapman & Hall/CRC.
4. Koller, D., & Friedman, N. (2009). Probabilistic graphical models: Principles and techniques. MIT Press.
5. Karlin, S., & Taylor, H. M. (1975). A first course in stochastic processes (2nd ed.). Academic Press.
6. Koller, D., & Friedman, N. (2009). Probabilistic graphical models: Principles and techniques. MIT Press.
7. Heckerman, D. (1995). Bayesian networks and Bayesian learning. In R. G. Cowell (Ed.), Probabilistic networks and expert systems (pp. 267-310). Springer.
8. Ross, S. M. (2010). Introduction to probability models (10th ed.). Academic Press.
9. Levin, D. A., Peres, Y., & Wilmer, E. L. (2009). Markov chains and mixing times. American Mathematical Society.
10. Murphy, K. P. (2002). Dynamic Bayesian networks: Representation, inference, and learning. PhD thesis, University of California, Berkeley.

APPENDIX

A.1 List of basic Python commands

1. Printing Output:
print(): Displays the value of an expression or variable to the console.
Python
print("Hello, world!")
2. Variables:
Assignment: Storing values in variables using the = operator.
Python
name = "Alice"
age = 30
Data Types: Python has various data types, including:
Numbers: Integers (int), floating-point numbers (float)
Strings: Sequences of characters enclosed in quotes (" or ').
Booleans: True or False.
Lists: Ordered collections of elements.
Tuples: Immutable ordered collections.
Dictionaries: Unordered key-value pairs.
3. Operators:
Arithmetic: +, -, *, /, //, %, **.
Comparison: ==, !=, <, >, <=, >=.
Logical: and, or, not.
4. Control Flow:
if statements: Execute code based on a condition.
Python
if age >= 18:
    print("You are an adult.")
else:
    print("You are a minor.")
for loops: Iterate over a sequence.
Python
for i in range(5):
    print(i)
while loops: Repeat code as long as a condition is true.
Python
count = 0
while count < 5:
    print(count)
    count += 1
5. Functions:
Defining functions: Creating reusable blocks of code.
Python
def greet(name):
    print("Hello, " + name + "!")

greet("Bob")
Calling functions: Executing a function with arguments.

6. Modules:
Importing modules: Accessing code from external files.
Python
```
import math

result = math.sqrt(25)
print(result)
```
7. Input and Output:
input(): Getting user input from the console.
Python
```
name = input("Enter your name: ")
print("Hello, " + name + "!")
```

A.2 Essential NumPy Commands: A Quick Reference

NumPy, a fundamental library for scientific computing in Python, provides a powerful array object and a large collection of high-level mathematical functions to operate on arrays. Here are some of the most commonly used commands:

Array Creation

np.array(list): Creates a NumPy array from a Python list.
np.arange(start, stop, step): Creates an array of evenly spaced values within a given interval.
np.linspace(start, stop, num): Creates an array of evenly spaced values over a specified interval.
np.zeros(shape): Creates an array of zeros with a given shape.
np.ones(shape): Creates an array of ones with a given shape.
np.random.rand(d0, d1, ...): Creates an array of random samples from a uniform distribution.
np.random.randn(d0, d1, ...): Creates an array of random samples from a standard normal distribution.

Array Attributes and Operations

array.shape: Returns the shape of the array (tuple of integers).
array.dtype: Returns the data type of the array elements.
array.ndim: Returns the number of dimensions of the array.
array.size: Returns the total number of elements in the array.
array.itemsize: Returns the size of each element in the array in bytes.
array.reshape(newshape): Reshapes the array without changing its data.
array.T: Transposes the array.
np.concatenate((array1, array2, ...)): Joins arrays along a specified axis.
np.vstack((array1, array2, ...)): Stacks arrays vertically.
np.hstack((array1, array2, ...)): Stacks arrays horizontally.

Mathematical Operations

np.add(array1, array2): Element-wise addition.
np.subtract(array1, array2): Element-wise subtraction.
np.multiply(array1, array2): Element-wise multiplication.
np.divide(array1, array2): Element-wise division.
np.power(array1, array2): Element-wise exponentiation.
np.sqrt(array): Element-wise square root.
np.sin(array): Element-wise sine.
np.cos(array): Element-wise cosine.
np.tan(array): Element-wise tangent.
np.exp(array): Element-wise exponential.
np.log(array): Element-wise natural logarithm.

Linear Algebra

np.dot(a, b): Matrix or dot product of two arrays.
np.linalg.inv(a): Inverse of a square matrix.
np.linalg.det(a): Determinant of a square matrix.
np.linalg.eig(a): Eigenvalues and eigenvectors of a square matrix.

Statistical Functions

np.mean(array): Calculates the mean of the array.
np.median(array): Calculates the median of the array.
np.std(array): Calculates the standard deviation of the array.
np.var(array): Calculates the variance of the array.
np.min(array): Finds the minimum value in the array.
np.max(array): Finds the maximum value in the array.

Basic formulas of probabilistic modeling

A3. Probability Basics:

Probability of an event: P(A) = (Number of favorable outcomes) / (Total number of possible outcomes)
Conditional probability: P(A|B) = P(A and B) / P(B)
Law of total probability: P(A) = Σ P(B_i) * P(A|B_i)
Probability Distributions:
Bernoulli distribution: P(X = k) = p^k * (1-p)^(1-k)
Binomial distribution: P(X = k) = C(n, k) * p^k * (1-p)^(n-k)
Poisson distribution: P(X = k) = (λ^k * e^(-λ)) / k!
Normal (Gaussian) distribution: f(x) = (1 / √(2πσ²)) * e^(-(x-μ)² / (2σ²))
Exponential distribution: f(x) = λ * e^(-λx)
Uniform distribution: f(x) = 1 / (b - a) for a ≤ x ≤ b
Statistical Moments:
Mean (expected value): E(X) = Σ x * P(X = x)
Variance: Var(X) = E((X - μ)²)
Standard deviation: σ = √Var(X)
Bayes' Theorem:
P(A|B) = P(B|A) * P(A) / P(B)
Markov Chains:
Transition probability matrix: P_ij = P(X_(n+1) = j | X_n = i)
Steady-state probability distribution: π = πP

A4. Installing and running Python

Python version 3.9.11
https://www.python.org/downloads/
https://code.visualstudio.com/

Instructions

1. Download and run python
2. Download and run visual studio
3. In visual studio, go to extensions
4. Select and install python (microsoft)
5. Create a new file
6. Name it test.py
7. Write a code
8. Press Run
9. Get results on the terminal